

Clean Code - Professionelle Codeerstellung und Wartung

Gesamtinhaltsverzeichnis

1	Clean Code – Einführung	1-3
1.1	Einleitung.....	1-3
1.2	Professioneller Code	1-4
1.2.1	Die Nutzer-Sicht.....	1-4
1.2.2	Die „Hacker“-Sicht.....	1-6
1.2.3	Die Programmierer-Sicht	1-7
1.2.4	Weitere Sichten	1-8
1.3	Fazit.....	1-10
1.4	„Code Smell“ – Anzeichen für schlechten Code	1-12
1.4.1	Verbreitete Smells.....	1-13
1.4.2	Weitere Smells und Programmierungs-Anti-Pattern	1-15
1.4.3	Architektur-Smells.....	1-16
1.5	Was ist professioneller Code?	1-17
1.6	Regeln	1-18
1.7	Technik.....	1-18
2	Objektorientierte Programmierung.....	2-3
2.1	Einleitung.....	2-3
2.2	Arten der Programmierung	2-4
2.3	Grundsätze objektorientierter Programmierung.....	2-7
2.3.1	Unified Modeling Language (UML)	2-7
2.3.2	Abstraktion.....	2-8
2.3.3	Klassen	2-8
2.3.4	Assoziation	2-10
2.3.5	Aggregation und Komposition.....	2-11
2.3.6	Komponenten	2-13
2.3.7	Vererbung.....	2-14
2.3.8	Polymorphie.....	2-15
2.3.9	Sichtbarkeiten	2-16
2.3.10	Nachrichten.....	2-18
2.4	Die drei „K“ der Objektorientierung	2-19
2.4.1	Kapselung (encapsulation)	2-19
2.4.2	Kopplung (coupling)	2-24
2.4.3	Kohäsion (cohesion)	2-28
2.5	Zusammenfassung	2-30

3	Professionelle Klassen und Objekte	3-3
3.1	Einleitung.....	3-3
3.2	Klassenhierarchien	3-4
3.2.1	Das Liskovsche Substitutionsprinzip (LSP).....	3-6
3.2.2	Die Holper-Regel	3-7
3.2.3	Mehrfachvererbungen.....	3-8
3.3	Schnittstellen	3-10
3.3.1	Hierarchieschnittstellen.....	3-10
3.3.2	Fähigkeitsschnittstellen.....	3-12
3.3.3	Mix-Ins	3-15
3.3.4	Parallele Schnittstellen-Hierarchien	3-16
3.3.5	Client-Schnittstellen	3-18
3.3.6	Das Interface-Segregation-Principle ISP	3-18
3.4	Klassengrößen	3-22
3.4.1	Das Visions-Prinzip.....	3-24
3.4.2	Das Single-Responsibility-Principle – SRP	3-25
3.5	Änderungen ermöglichen	3-27
3.5.1	Das Open-Closed-Principle – OCP)	3-27
3.5.2	Das Dependency-Inversion-Principle – DIP).....	3-29
3.6	Zusammenfassung	3-31
4	Namen	4-3
4.1	Einleitung.....	4-3
4.2	Welche Sprache?	4-3
4.3	Bedeutungsvolle Namen	4-5
4.3.1	Klassen	4-7
4.3.2	Abstrakte Klassen	4-7
4.3.3	Interfaces	4-8
4.3.4	Methoden.....	4-9
4.3.5	Konstruktoren	4-10
4.3.6	Namen und Kontexte	4-11
4.3.7	Besondere Namen	4-12
4.3.8	Missverständliche Namen.....	4-16
4.3.9	Textrauschen	4-17
4.3.10	Domänen-Sprache vs. Lösung-Sprache	4-19
4.3.11	Ein Konzept, ein Wort	4-19
4.3.12	Verwandte Konzepte.....	4-20
4.4	Namen und ihre Form.....	4-21
4.4.1	Groß- und Kleinschreibung	4-21
4.4.2	Optische Verwechslungen	4-22

4.4.3	Aussprechbare Namen	4-22
4.4.4	Typ- und Kontextbezeichner (encodings)	4-24
4.4.5	Wortspiele und „Slang“	4-26
4.5	Vorgehen	4-27
4.5.1	Ändern von Namen	4-27
4.5.2	Der Style-Guide	4-28
4.6	Zusammenfassung	4-29
5	Methoden	5-3
5.1	Einleitung	5-3
5.1.1	Was ist eine Methode?	5-3
5.2	Form	5-5
5.2.1	Länge	5-6
5.2.2	Blockgrößen	5-12
5.2.3	Namen	5-15
5.3	Inhalt	5-17
5.3.1	Eine Aufgabe	5-17
5.3.2	Die Vision	5-19
5.3.3	Abstraktionsebenen	5-19
5.3.4	Die Stepdown-Regel	5-19
5.4	Argumente	5-21
5.4.1	Niladische Methoden	5-22
5.4.2	Monadische Methoden	5-22
5.4.3	Dyadische Methoden	5-24
5.4.4	Triadische Methoden	5-25
5.4.5	Größere (Polyadische) Methoden	5-26
5.4.6	Flags	5-27
5.4.7	Ausgabe Parameter	5-29
5.4.8	Argument-Objekte	5-30
5.5	Stil	5-32
5.5.1	Seiteneffekte	5-32
5.5.2	Befehl oder Abfrage (Command Query Separation)	5-32
5.5.3	Mehrere Exit-Punkte	5-33
5.5.4	Rekursionen	5-34
5.6	Zusammenfassung	5-36
6	Kommentare und Dokumentation	6-3
6.1	Einleitung	6-3
6.1.1	Lesbarer Code	6-4
6.2	Gute Kommentare	6-6
6.2.1	Rechtliche Hinweise	6-7

6.2.2	Klarstellungen	6-8
6.2.3	Absichtserklärungen	6-8
6.2.4	Design Patterns	6-8
6.2.5	Regelverstöße	6-9
6.2.6	Unterstreichungen	6-9
6.2.7	Formale Kommentare	6-9
6.3	Schlechte Kommentare	6-12
6.3.1	Unverständliche Kommentare.....	6-12
6.3.2	Redundanzen	6-13
6.3.3	Forcierte Kommentare	6-13
6.3.4	Codehistorien.....	6-14
6.3.5	Klammer-Kommentare.....	6-15
6.3.6	Auskommentierter Code	6-16
6.3.7	Informationsüberfluss.....	6-16
6.3.8	TODOs.....	6-17
6.3.9	Nicht-öffentliche formale Kommentare.....	6-18
6.4	Testfälle als Dokumentation	6-19
6.5	Zusammenfassung	6-20
7	Code-Formatierung.....	7-3
7.1	Einleitung.....	7-3
7.2	Warum Formatierung.....	7-3
7.2.1	Automatisierte Formatierung.....	7-4
7.2.2	Sourcecode als Kommunikation	7-6
7.3	Die Zeitungsmetapher	7-7
7.3.1	Schlagzeile	7-8
7.3.2	Untertitel	7-8
7.3.3	Der Einstieg / Lead	7-9
7.3.4	Absätze.....	7-9
7.3.5	Reihenfolgen.....	7-12
7.3.6	Die Rubrik	7-13
7.4	Weitere Formatierungsregeln	7-14
7.4.1	Breite und Höhe.....	7-14
7.4.2	Einrückungen.....	7-15
7.4.3	Ausnahmen.....	7-16
7.5	Team Rules!	7-17
7.6	Zusammenfassung	7-18
8	Clean Code – Spezielle Themen	8-3
8.1	Einleitung.....	8-3
8.2	Exception Handling.....	8-4

8.2.1	Die Verwendung von Null	8-8
8.3	Nebenläufigkeit.....	8-9
8.3.1	Mythen und Missverständnisse.....	8-10
8.3.2	Die Herausforderung.....	8-12
8.3.3	Nebenläufige Prinzipien	8-14
8.3.4	Garbage Collector.....	8-17
8.3.5	Zusammenfassung	8-18
8.4	Optimierung	8-19
8.4.1	Einleitung	8-19
8.4.2	Was ist Performance?.....	8-19
8.4.3	Gefühlte Performance	8-20
8.4.4	Wann sollte optimiert werden?.....	8-21
8.4.5	Das Optimierungsdreieck.....	8-22
8.4.6	Zusammenfassung	8-23
9	Metriken	9-3
9.1	Einleitung.....	9-3
9.1.1	Code Entropy	9-3
9.1.2	Die Zeitachse	9-3
9.2	Basis Metriken	9-4
9.2.1	Cyclomatic Complexity (CC)	9-4
9.2.2	Lines of Code (LOC)	9-5
9.2.3	Non Commenting Source Statements (NCSS)	9-6
9.3	Objektorientierte Metriken	9-7
9.3.1	Weighted Methods per Class (WMC).....	9-7
9.3.2	Depth of Inheritance Tree (DIT)	9-7
9.3.3	Number of Children (NOC)	9-7
9.3.4	Coupling between Object Classes (CBO)	9-7
9.3.5	Response for a Class (RFC)	9-8
9.3.6	Lack of Cohesion in Methods (LCOM)	9-8
9.3.7	Bewertung.....	9-9
9.4	Statische Analyse Tools (Bug Finder)	9-10
9.5	Laufzeit Metriken	9-11
9.5.1	Testabdeckung	9-11
9.5.2	Builddauer.....	9-11
9.6	Zusammenfassung	9-12
10	Literaturempfehlungen	3
	Gesamtindex.....	IDX-1

1


Clean Code – Einführung

1.1	Einleitung.....	1-3
1.2	Professioneller Code	1-4
1.2.1	Die Nutzer-Sicht.....	1-4
1.2.2	Die „Hacker“-Sicht.....	1-6
1.2.3	Die Programmierer-Sicht	1-7
1.2.4	Weitere Sichten	1-8
1.3	Fazit.....	1-10
1.4	„Code Smell“ – Anzeichen für schlechten Code	1-12
1.4.1	Verbreitete Smells.....	1-13
1.4.2	Weitere Smells und Programmierungs-Anti-Pattern	1-15
1.4.3	Architektur-Smells.....	1-16
1.5	Was ist professioneller Code?	1-17
1.6	Regeln	1-18
1.7	Technik.....	1-18

1 Clean Code – Einführung

1.1 Einleitung

Im folgenden Kapitel wollen wir einige Grundlagen definieren und abgrenzen was Clean Code ist und woher die Prinzipien stammen.

Clean Code	
<ul style="list-style-type: none">▪ Begriff geht zurück auf ein Buch von Robert C. Martin<ul style="list-style-type: none">▪ Konzepte und Ideen stammen aus dem Software-Engineering▪ Sind eigentlich altbekannt ▪ Zusammenfassung von Konzepten und Ideen aus dem Software-Engineering<ul style="list-style-type: none">▪ Mit den OO Sprachen und objektorientierten Software-Entwicklung entstanden▪ Im Wesentlichen geht's um:<ul style="list-style-type: none">▪ Saubere Strukturen<ul style="list-style-type: none">▪ Siehe Design Pattern▪ Namenskonventionen▪ Klassen- und Methoden-Größen<ul style="list-style-type: none">▪ Siehe Smalltalk ▪ Lässt sich auf Scriptsprachen und non OO-Sprachen nur begrenzt anwenden	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
1 Seite 2	

„**Clean Code** (Quelle Wikipedia) ist ein Begriff aus der [Softwaretechnik](#), der seinen Ursprung im gleichnamigen Buch von [Robert Cecil Martin](#) hat. Als „sauber“ bezeichnen [Softwareentwickler](#) in erster Linie Quellcode, aber auch Dokumente, Konzepte, Regeln und Verfahren, die intuitiv verständlich sind. Als intuitiv verständlich gilt alles, was mit wenig Aufwand und in kurzer Zeit richtig verstanden werden kann. Vorteile von Clean Code sind stabilere und effizient wartbarere Programme, d. h. kürzere Entwicklungszeiten bei Funktionserweiterung und Fehlerbehebungen. Die Bedeutung wächst mit der Beobachtung, dass im Schnitt 80 % der Lebensdauer einer Software auf den Wartungszeitraum entfällt.

Schwierigkeiten beim Entwickeln von Clean Code liegen häufig in zunächst unklaren oder sich widersprechenden Anforderungen, zum Teil begründet im Fehlen von Erfahrung im Entwickeln von Clean Code, im Mangel an Disziplin beim Programmieren und im Aufwand nachträglicher [Quellcode](#)-Bereinigungen (dem sog. [Refactoring](#)).

Die Notwendigkeit, Code noch nach der Entwicklung von „unsauberen“ Stellen zu reinigen, wird häufig nicht gesehen oder vom Management nicht bewilligt, sobald das Programm seine vorgesehene Funktion ausübt. Ein direktes Schreiben von „sauberen“ Code ist nahezu unmöglich, kann jedoch durch den bewussten Umgang mit den Prinzipien und Praktiken von Clean Code verbessert werden.

Eng verbunden mit dem Begriff *Clean Code* sind Maßnahmen, die bei der Entwicklung von Software zu „sauberen“ Programmcode führen. So zahlreich wie die Gründe für „unsauberen“ Code sind, so vielfältig sind auch die vorgeschlagenen Regeln in den aufgestellten Maßnahmenkatalogen. Dazu gehören:

[Quelltextformatierung](#) (engl. code conventions),

[Entwurfsmuster](#) (engl. design patterns),

[Konvention vor Konfiguration](#) (engl. convention over configuration),

eine umfangreiche Menge an Vorschlägen aus dem Buch *Clean Code* von Robert C. Martin.“

1.2 Professioneller Code


Unser anspruchsvolles Ziel lautet, professionellen Code zu erstellen. Um es erreichen zu können, müssen wir zunächst natürlich erst einmal definieren, was professioneller Code überhaupt ist. Die Meinungen hierüber gehen wie zu erwarten deutlich auseinander.

Um bei Definition einen Schritt weiter zu kommen, wollen wir zunächst den Begriff „Professionell“ durch eine adäquate Alternative ersetzen: „qualitativ hochwertig“. Wir wollen also eine möglichst hohe Qualität erreichen. Der Begriff Qualität scheint zumindest im ersten Ansatz besser zu definieren zu sein.

Allerdings gibt es natürlich auch bei dem Begriff Qualität unterschiedliche Sichtweisen. Wir wollen im Folgenden die wichtigsten davon näher betrachten und uns diejenigen herausuchen, die uns am geeignetsten erscheinen, um unser Ziel zu formulieren.

1.2.1 Die Nutzer-Sicht

Für einen Nutzer ist natürlich in erster Linie die Funktionsfähigkeit des Programmes selbst entscheidend. Weiterhin ist relevant, wie benutzerfreundlich oder effizient das Programm ist. Steve McConnell hat die für den Nutzer wichtigen Kriterien in seinem Buch „Code Complete“ als „Externe Softwarequalitäts-Merkmale“ definiert:

<div>Die Nutzer-Sicht (Externe Qualitätsmerkmale) </div> <div><ul style="list-style-type: none">▪ Nach Steve McConnell „Code Complete“ als „Externe Softwarequalitäts-Merkmale“ definiert:<ul style="list-style-type: none">▪ Korrektheit▪ Benutzerfreundlichkeit▪ Effizienz▪ Zuverlässigkeit▪ Integrität▪ Anpassungsfähigkeit▪ Genauigkeit▪ Robustheit</div>
<div>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH</div> <div>Clean Code – Professionelle Codeerstellung und Wartung</div> <div>1 Seite 3</div>

Korrektheit: Der Grad der Fehlerfreiheit eines System (bezogen auf Spezifikation, Entwurf und Implementierung). Wie korrekt erfüllt das Programm seine Aufgabe?

Benutzerfreundlichkeit: Wie leicht fällt es dem Benutzer, das System zu erlernen und zu benutzen?

Effizienz: Wie gut werden vorhandene Ressourcen genutzt? Welchen Speicher/Leistungsbedarf besitzt das Programm? Wie *schnell* ist es?

Zuverlässigkeit: Wie groß ist die Wahrscheinlichkeit, dass das Programm unter normalen, definierten Bedingungen ausfällt? Wie hoch ist die damit verbundene Ausfallzeit?

Integrität: Wie sicher und stabil sind die Daten? Sind die Daten immer konsistent (Transaktionalität, Nebenläufigkeit)? Existieren Maßnahmen, um einen unauthorisierten Zugriff auf die Daten zu verhindern?

Anpassungsfähigkeit: Wie flexibel kann das Programm ohne Änderung an die Umgebung oder andere Programme angepasst werden (= Konfigurierbarkeit)

Genauigkeit: Wie genau sind die Ergebnisse (nicht: *sind die Ergebnisse richtig*)? Die Auflösung der Ergebnisse.


Robustheit: Wie geht das Programm mit externen Fehlersituationen um (Netzwerkausfall, Fehleingaben)? Wie häufig führen diese zu einem Ausfall des Programms?

Natürlich gibt es bei diesen Merkmalen einige, die sich nur im Detail unterscheiden.

Wir werden externe Softwarequalität im Weiteren auch als *Produkt-Qualität* bezeichnen. Wo hier in der Entwicklung die Schwerpunkte liegen, wird durch die Anforderungen beschrieben – Korrektheit ist natürlich immer ein entscheidendes Kriterium (also die Funktionalität), die anderen Merkmale werden durch die sogenannten Nicht-Funktionalen Anforderungen (NFAs) beschrieben.

1.2.2 Die „Hacker“-Sicht

Eine Sicht, die für uns nicht weiter von Bedeutung ist, aber der Vollständigkeit halber dennoch erwähnt werden sollte. Diese Sichtweise beschreibt ein Programm bzw. ein Stück Software dann als professionell¹, wenn das Problem möglichst „innovativ“ (soll heißen: möglichst überraschend) gelöst wurde. Die Hacker-Sicht dient dazu, die eigenen Fähigkeiten unter Beweis zu stellen oder sich für eine Firma unabdingbar zu machen.

Die Hacker-Sicht	 integrata cegos
<ul style="list-style-type: none">▪ Nicht weiter von Bedeutung▪ Sichtweise beschreibt ein Programm bzw. ein Stück Software dann als professionell, wenn das Problem möglichst „innovativ“ (soll heißen: möglichst überraschend) gelöst wurde.▪ Die Hacker-Sicht dient dazu, die eigenen Fähigkeiten unter Beweis zu stellen oder sich für eine Firma unabdingbar zu machen.▪ Leider wird die hieraus entstehende Herangehensweise immer noch häufig in der Praxis beobachtet▪ Aber nicht unbedingt als qualitativ hochwertig	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
1 Seite 4	

Leider wird die hieraus entstehende Herangehensweise immer noch häufig in der Praxis beobachtet.

¹ Aber nicht unbedingt als qualitativ hochwertig

1.2.3 Die Programmierer-Sicht

Die Programmierer-Sicht misst die Qualität eines Programmes anhand von Kriterien, die für den Nutzer nicht direkt sichtbar sind, wie beispielsweise Lesbarkeit des Quellcodes, und deshalb analog zu den externen als interne Qualitätsmerkmale bezeichnet werden. Ein anderer gängiger Begriff ist „Sourcecode-Qualität“². Grob betrachtet beeinflussen diese Merkmale alle den Aufwand, den ein Programmierer zukünftig erbringen muss, um bestimmte Ziele zu erreichen.

Die Programmierer-Sicht (interne Qualitätsmerkmale)	
<ul style="list-style-type: none">▪ Wartungsfreundlichkeit▪ Flexibilität▪ Portierbarkeit▪ Wiederverwendbarkeit▪ Lesbarkeit▪ Testbarkeit▪ Verständlichkeit	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
1 Seite 5	

Die wichtigsten Merkmale sind:

Wartungsfreundlichkeit: Wie leicht kann der Code angepasst werden, um Fehler zu korrigieren, die Leistungsfähigkeit zu erhöhen oder Fähigkeiten geändert bzw. hinzugefügt werden?

Flexibilität: Wie leicht lässt sich das Programm an neue Situationen anpassen?

Portierbarkeit: Wie viel Arbeit muss aufgewendet werden, um das Programm in einer anderen Umgebung einzusetzen?

Wiederverwendbarkeit: Können Teile des Codes in anderen Programmen/Systemen verwendet werden?³

² Eine weitere, etwas verkürzende Bezeichnung ist „Les- und Wartbarkeit“. Hierbei werden allerdings einige Punkte nicht gewürdigt.

³ Das heißt nicht, dass der Code jemals in anderen Systemen verwendet wird. Dennoch ist das Maß der Wiederverwendbarkeit ein deutlicher Prüfstein für die Sourcecode-Qualität

Lesbarkeit: Wie viel Anstrengung muss ein Entwickler aufbringen, um den Code zu verstehen? Wie viel Fachwissen muss dieser dazu mitbringen?


Testbarkeit: Wie gut lässt sich das Programm bzw. einzelne Komponenten (automatisiert) testen?

Verständlichkeit: Wie leicht ist das Programm an sich verständlich – also wie die Komponenten zusammenarbeiten, bzw. „was das Programm macht“?

Bei den internen Merkmalen gibt es ebenso wie bei den externen Merkmalen natürlich Überschneidungen bzw. die Merkmale selbst unterscheiden sich teilweise nur in Nuancen. Auch wirken einige Merkmale sicher in den anderen Bereich hinein. Mit der Frage, welche Merkmale wie zusammen hängen werden wir uns im Folgenden noch ein wenig ausführlicher beschäftigen.

Wie wir dabei auch sehen werden, nehmen zwei Merkmale, nämlich Flexibilität und Portierbarkeit eine Sonderstellung ein.

1.2.4 Weitere Sichten

Weitere Sichten	
<ul style="list-style-type: none">▪ Die betriebswirtschaftliche Sichtweise<ul style="list-style-type: none">▪ Qualität = Kosten<ul style="list-style-type: none">▪ Erstellung eines Programmes▪ Wartung, Pflege und Erweiterung▪ Wartung eines Programmes vielfaches an Kosten gegenüber Entwicklung▪ Die Auftraggeber-Sicht<ul style="list-style-type: none">▪ vollkommene Umkehrung▪ internen Merkmale nebensächlich▪ Kosten, Dauer und externe Merkmale wichtig	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung 1 Seite 6

Die betriebswirtschaftliche Sicht

Die betriebswirtschaftliche Sichtweise verfolgt eine gänzlich andere Strategie. Qualität bezeichnet hierbei die Kosten die zum einen für die Erstellung eines Programmes, zum anderen aber auch im weiteren Leben der Software für Wartung, Pflege und Erweiterung anfallen. Ein aus betriebswirtschaftlicher Sicht hochwertiges Programm kostet also in Entwicklung und Pflege möglichst wenig.

Traditionell kostet die Wartung eines Programmes ein Vielfaches der Entwicklungskosten, weshalb die betriebswirtschaftliche Sicht natürlich Herangehensweisen favorisiert (bzw. favorisieren sollte), die möglichst den Wartungsaufwand reduzieren.

Es gilt der Grundsatz: „Gerade so viel nötig“, was die externen Merkmale angeht. Also nicht maximale Effizienz oder Benutzerfreundlichkeit, sondern eben nur so viel, wie gefordert wurde (anhand des Pflichtenheftes bzw. anderer Anforderungsdokumente).

Man könnte die betriebswirtschaftliche Sicht auch als Zukunftsorientierte Sicht betrachten.


Natürlich soll nicht unerwähnt bleiben, dass zum einen die Weitsicht für diese Sicht in der Praxis fehlt und zum anderen, dass in besonderen Fällen die Gewichtung aus betriebswirtschaftlicher Sicht gänzlich anders sein kann. Das ist zum Beispiel der Fall, wenn das Produkt explizit nicht gepflegt wird, sondern lediglich als Wegwerfprodukt in einem klar definierten Rahmen genutzt wird (zum Beispiel zu Konvertierung eines Datenbestandes): Allerdings hat die Erfahrung immer wieder gezeigt, dass auch Code, der nie für eine längere Nutzung gedacht war, doch über Jahre hinweg nicht nur genutzt, sondern eben auch gepflegt und weiter entwickelt wird.

Die Auftraggeber-Sicht

Die vollkommene Umkehrung der betriebswirtschaftlichen Sicht ist die Auftraggeber-Sicht. Für den Auftraggeber (nicht den späteren Nutzer) ist es weniger relevant, wie gut eine Software tatsächlich funktioniert, sondern dass sie die *geforderten Kriterien erfüllt*. Für diese Sichtweise sind die internen Merkmale nebensächlich. Stattdessen ist ein Maßgebliches Kriterium für Qualität (bzw. hier natürlich auch eher von Professionalität), wie lange die Auftragserfüllung dauert.

1.3 Fazit

Das Ganze kann natürlich nur eine exemplarische Aufstellung sein, die genauen Einflüsse unterscheiden sich von Projekt zu Projekt.

Fazit	
<ul style="list-style-type: none">▪ Eine maximale, externe Qualität ist nicht erreichbar, Schwerpunkte müssen anhand klarer Anforderungen gestellt werden.▪ Eine Betonung eines externen Merkmal muss durch Nicht-funktionale Anforderungen (NFAs) begründet sein.▪ <i>Wartungsfreundlichkeit, Wiederverwendbarkeit, Lesbarkeit und Testbarkeit</i> sollten, solange der Aufwand vertretbar ist, so hoch wie möglich sein. (Siehe Regeln im Skript)▪ Die Programmierer-Sicht<ul style="list-style-type: none">▪ <i>Intrinsische</i> Qualitätsmerkmale<ul style="list-style-type: none">▪ Wartungsfreundlichkeit▪ Wiederverwendbarkeit▪ Lesbarkeit▪ Testbarkeit▪ Verständlichkeit▪ <i>Extrinsische</i> Qualitätsmerkmale<ul style="list-style-type: none">▪ Flexibilität▪ Portierbarkeit	
<small>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH</small>	<small>1 Seite 7</small>

Verbesserungen der Benutzerfreundlichkeit ziehen in der Regel eine Verschlechterung der Sourcecode-Qualität nach sich. Das liegt daran, dass eine benutzerfreundliche Oberfläche in der Regel viele verschiedene, intelligente Möglichkeiten anbietet, um dasselbe Ziel zu erreichen und damit den Code mit sehr vielen Fallunterscheidungen anreichert, was sich insbesondere deutlich negativ auf die Les- und Testbarkeit auswirkt.

Viele Verbesserungen an den externen Merkmalen bewirken potentiell eine Verschlechterung der Lesbarkeit, was sich ggf. wieder negativ auf andere Punkte auswirkt.


Verbesserungen an Wartungsfreundlichkeit, Wiederverwendbarkeit, Lesbarkeit und Testbarkeit (und in geringerem Maße auch Verständlichkeit) haben in der Regel positive Auswirkungen auf eine Vielzahl anderer (insbesondere interner, aber auch externer) Merkmale. Gleichzeitig ziehen diese Verbesserungen keine negativen Konsequenzen nach sich.

Extrinsische und Intrinsische Merkmale

Um diese Unterscheidung auch im Sprachgebrauch deutlich zu machen, unterscheiden wir die internen Merkmale weiterhin in *intrinsische* und *extrinsische Merkmale*. Intrinsische sind dabei diejenigen Merkmale, die sich gegenseitig bedingen, aber nach außen hin wenig Auswirkungen haben (also die eigentliche Sourcecode-Qualität), konkret: Wartungsfreundlichkeit, Wiederverwendbarkeit, Lesbarkeit, Testbarkeit und Verständlichkeit, die beiden extrinsischen dagegen diejenigen, die eben eine konkrete, von außen vorgegebene Forderung erfüllen.

Die beiden extrinsischen Merkmale (Flexibilität und Portierbarkeit) nehmen also eigentlich eine Zwitterstellung zwischen externen und internen Merkmalen ein. Zum einen könnten sie als extern betrachtet werden, weil sie durch äußere Anforderungen begründet werden, zum anderen handelt es sich aber um Eigenschaften, die zu allererst direkt den Programmierer betreffen.

1.4 „Code Smell“⁴ – Anzeichen für schlechten Code

Code Smell	
<ul style="list-style-type: none"> ▪ Code-Smell, kurz Smell (engl. ‚[schlechter] Geruch‘) oder deutsch übelriechender Code versteht man in der Programmierung ein Konstrukt, das eine Überarbeitung des Programm-Quelltextes nahelegt. Dem Vernehmen nach stammt die Metapher <i>Smell</i> von Kent Beck und erlangte weite Verbreitung durch das Buch <i>Refactoring</i> von Martin Fowler. ▪ Begriff soll handfestere Kriterien für Refactoring beschreiben, als vagen Hinweis auf Programmästhetik ▪ Code-Smell sind keine Programmfehler ▪ Funktionierender Programmcode, schlecht strukturiert ▪ Code für Programmierer schwer verständlich ▪ Bei Korrekturen und Erweiterungen schleichen sich häufig wieder neue Fehler ein ▪ Code-Smell kann Hinweis auf ein tieferes Problem sein <ul style="list-style-type: none"> ▪ Verborgene schlechte Struktur, die erst durch eine Überarbeitung erkannt wird 	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung 1 Seite 8

Auszug aus Wikipedia:


„Unter **Code-Smell**, kurz **Smell** (engl. ‚[schlechter] Geruch‘) oder deutsch **übelriechender Code** versteht man in der Programmierung ein Konstrukt, das eine Überarbeitung des Programm-Quelltextes nahelegt. Dem Vernehmen nach stammt die Metapher *Smell* von Kent Beck und erlangte weite Verbreitung durch das Buch *Refactoring* von Martin Fowler. Unter dem Begriff sollten handfestere Kriterien für Refactoring beschrieben werden, als das durch den vagen Hinweis auf Programmästhetik geschehen würde.

Bei Code-Smell geht es nicht um Programmfehler, sondern um funktionierenden Programmcode, der aber schlecht strukturiert ist. Das größte Problem liegt darin, dass der Code für den Programmierer schwer verständlich ist, so dass sich bei Korrekturen und Erweiterungen häufig wieder neue Fehler einschleichen. Code-Smell kann auch auf ein tieferes Problem hinweisen, das in der schlechten Struktur verborgen liegt und erst durch eine Überarbeitung erkannt wird.“

⁴ Quelle Wikipedia.

1.4.1 Verbreitete Smells

Die folgenden von Martin Fowler und Kent Beck beschriebenen Smells beziehen sich auf die objektorientierte Programmierung, haben aber naheliegende Entsprechungen unter anderen Programmier-Paradigmen siehe <https://de.wikipedia.org/wiki/Code-Smell>

Verbreitete Code Smells (1)	
<ul style="list-style-type: none">▪ Code-Duplizierung<ul style="list-style-type: none">▪ Der gleiche Code kommt an verschiedenen Stellen vor.▪ Lange Methode<ul style="list-style-type: none">▪ Eine Methode (Funktion, Prozedur) ist zu lang.▪ Große Klasse<ul style="list-style-type: none">▪ Eine Klasse ist zu umfangreich, umfasst zu viele Instanzvariablen und duplizierten Code. Siehe auch Gottobjekt.▪ Lange Parameterliste<ul style="list-style-type: none">▪ Anstatt ein Objekt an eine Methode zu übergeben, werden Objekt-Attribute extrahiert und der Methode als lange Parameterliste übergeben.▪ Divergierende Änderungen<ul style="list-style-type: none">▪ Für eine Änderung muss eine Klasse an mehreren Stellen angepasst werden.▪ Schrotkugeln herausoperieren (engl. Shotgun Surgery)<ul style="list-style-type: none">▪ Dieser Smell ist noch gravierender als divergierende Änderungen: Für eine Änderung müssen weitere Änderungen an vielen Klassen durchgeführt werden.▪ Neid (engl. Feature Envy)<ul style="list-style-type: none">▪ Eine Methode interessiert sich mehr für die Eigenschaften – insbesondere die Daten – einer anderen Klasse als für jene ihrer eigenen Klasse.	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
	1 Seite 9

Diese Aufstellung ist interessanterweise ähnlich dem, was im Buch Clean Code behandelt wird. Diese Smells sind zusammengetragen aus verschiedenen Erfahrungen und eigentlich eher allgemeingültig.

Verbreitete Code Smells (2)



- Datenklumpen
 - Eine Gruppe von Objekten kommt häufig zusammen vor: als Felder in einigen Klassen und als Parameter vieler Methoden.
- Neigung zu elementaren Typen (engl. Primitive Obsession)
 - Elementare Typen werden benutzt, obwohl auch für einfache Aufgaben Klassen und Objekte aussagekräftiger sind.
- Case-Anweisungen in objektorientiertem Code
 - Switch-Case-Anweisungen werden benutzt, obwohl Polymorphismus sie weitgehend überflüssig macht und das damit zusammenhängende Problem des duplizierten Codes löst.
- Parallele Vererbungshierarchien
 - Zu jeder Unterklasse in der einen Hierarchie gibt es immer auch eine Unterklasse in einer anderen Hierarchie.
- Faule Klasse
 - Eine Klasse leistet zu wenig, um ihre Existenz zu rechtfertigen.
- Spekulative Allgemeinheit
 - Es wurden alle möglichen Spezialfälle vorgesehen, die gar nicht benötigt werden; solch allgemeiner Code braucht Aufwand in der Pflege, ohne dass er etwas nützt.

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code – Professionelle Codeerstellung und Wartung

1 Seite 10

Verbreitete Code Smells (3)




- Temporäre Felder
 - Ein Objekt verwendet eine Variable nur unter bestimmten Umständen – der Code ist schwer zu verstehen und zu debuggen, weil das Feld scheinbar nicht verwendet wird.
- Nachrichtenketten
 - Das Gesetz von Demeter wird verletzt.
- Middle Man (Vermittler)
 - Eine Klasse delegiert alle Methodenaufrufe an eine andere Klasse.
- Unangebrachte Intimität
 - Zwei Klassen haben zu enge Verflechtungen miteinander.
- Alternative Klassen mit verschiedenen Schnittstellen
 - Zwei Klassen machen das gleiche, verwenden hierfür aber unterschiedliche Schnittstellen.
- Inkomplette Bibliotheksklasse
 - Eine Klasse einer Programmbibliothek benötigt Erweiterungen, um in einem für sie geeigneten Bereich verwendet werden zu können.

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH


Clean Code – Professionelle Codeerstellung und Wartung

1 Seite 11

<h3>Verbreitete Code Smells (4)</h3>		
<ul style="list-style-type: none">▪ Datenklasse<ul style="list-style-type: none">▪ Klassen mit Feldern und Zugriffsmethoden ohne Funktionalität.▪ Ausgeschlagenes Erbe (engl. Refused Bequest)<ul style="list-style-type: none">▪ Unterklassen brauchen die Methoden und Daten gar nicht, die sie von den Oberklassen erben (siehe auch Liskovsches Substitutionsprinzip)▪ Kommentare<ul style="list-style-type: none">▪ Kommentare erleichtern im Allgemeinen die Verständlichkeit. Kommentare erscheinen jedoch häufig genau dort notwendig zu sein, wo der Code schlecht ist. Kommentare können somit ein Hinweis auf schlechten Code sein.		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	1 Seite 12

1.4.2 Weitere Smells und Programmierungs-Anti-Pattern

Neben den von Fowler erwähnte Smells gibt es noch eine Reihe von Code-Smells die oft auch unter [Programmierungs-Anti-Pattern](#) erwähnt werden:

<h3>Weitere Smells und Programmierungs-Anti-Pattern (1)</h3>		
<ul style="list-style-type: none">▪ Neben den von Fowler erwähnten Smells gibt es noch eine Reihe von Code-Smells, die oft auch unter Programmierungs-Anti-Pattern erwähnt werden:▪ Nichtssagender Name (engl. Uncommunicative Name)<ul style="list-style-type: none">▪ Name, der nichts über Eigenschaften oder Verwendung des Benannten aussagt. Aussagekräftige Namen sind wesentlich für das Verständnis von Programmcode.▪ Redundanter Code<ul style="list-style-type: none">▪ Ein Stück Code, das nicht (mehr) verwendet wird.▪ Exhibitionismus (engl. Indecent Exposure)<ul style="list-style-type: none">▪ Interne Details einer Klasse sind unnötigerweise Teil ihrer Schnittstelle nach außen.▪ Contrived complexity (engl.)<ul style="list-style-type: none">▪ Erzwungene Verwendung von Entwurfsmustern, wo einfacheres Design ausreichen würde.		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	1 Seite 13

Weitere Smells und Programmierungs-Anti-Pattern (2)



- Zu lange Namen
 - Insbesondere die Verwendung von Architektur- oder Designbestandteilen in den Namen von Klassen oder Methoden.
- Zu kurze Namen
 - Die Verwendung von „x“, „i“ oder Abkürzungen. Der Name einer Variable sollte ihre Funktion beschreiben.
- Über-Callback
 - Ein Callback, der versucht, alles zu tun.
- Komplexe Verzweigungen
 - Verzweigungen, die eine Menge von Bedingungen abprüfen, die mit der Funktionalität des Codeblocks nichts zu tun haben.
- Tiefe Verschachtelungen
 - Verschachtelte if/else/for/do/while-Statements. Sie machen den Code unlesbar.

1.4.3 Architektur-Smells

Neben den von Beck und Fowler adressierten Smells im Quelltext von Anwendungen treten Smells auch in der Architektur von Softwaresystemen auf. Diese wurden von Stefan Roock und Martin Lippert beschrieben.

Architektur Smells



- Weitere Gruppe der Smells
- Hierzu zählen unter anderem:
 - Zyklische Benutzungsbeziehungen zwischen Paketen, Schichten und Subsystemen
 - Größe und Aufteilung der Pakete oder Subsysteme
- Daraus resultieren Design Pattern
 - Layer
 - MVC z.B. mit der Observer
 - Etc.

1.5 Was ist professioneller Code?

Kehren wir also zurück zur Ausgangsfrage: „was ist professioneller Code?“

Für uns ist also professioneller Code Sourcecode, der die intrinsischen Qualitätsmerkmale (Wartungsfreundlichkeit, Wiederverwendbarkeit, Lesbarkeit, Testbarkeit und Verständlichkeit) maximiert, aber dabei natürlich die **erforderlichen** externen Qualitätsmerkmale umsetzt.

Im weiteren Verlauf dieser Unterlage werden wir uns mit den externen Qualitätsmerkmalen nur noch am Rande auseinandersetzen, da diese ja maßgeblich durch die NFAs bestimmt werden. Das Ziel, dass wir ab hier verfolgen heißt demnach, die intrinsischen Merkmale, also die Qualität unseres Sourcecodes so weit wie möglich zu steigern. Jede Maßnahme wollen wir dabei direkt an den intrinsischen Merkmalen, die sie beeinflusst, bewerten.

Wenn wir in Zukunft Regeln aufstellen, werden wir sie daher mit Kürzeln versehen, die die dazu gehörigen intrinsischen Merkmale beschreiben (Wart, Wied, Lesb, Test, Vers).

Zusammenfassung: Was ist professioneller Code?



- Code, der
 - die *Intrinsischen Qualitätsmerkmale* maximiert
 - Wartungsfreundlichkeit
 - Wiederverwendbarkeit
 - Lesbarkeit
 - Testbarkeit
 - Verständlichkeit
 - die **erforderlichen**, externen Qualitätsmerkmale umsetzt
- Code, der die Smells vermeidet
- Code, der die Architektur Muster berücksichtigt bzw. Architektur Smells vermeidet

Tatsächlich werden wir feststellen, dass dabei Lesbarkeit einen überproportional hohen Stellenwert bekommt. Bei genauerer Überlegung liegt das natürlich nahe. Um einen Code zu warten oder ihn wiederzuverwenden, muss er ja zwangsläufig erst gelesen und **verstanden** worden sein.

1.6 Regeln

Über diese Unterlage verstreut findet sich eine Reihe von fundamentalen Regeln.

1.7 Technik

Die Programmbeispiele in dieser Unterlage sind größtenteils in Java verfasst, die Konzepte gelten aber grundsätzlich für alle objektorientierten Programmiersprachen

2


Objektorientierte Programmierung

2.1	Einleitung.....	2-3
2.2	Arten der Programmierung	2-4
2.3	Grundsätze objektorientierter Programmierung.....	2-7
2.3.1	Unified Modeling Language (UML)	2-7
2.3.2	Abstraktion.....	2-8
2.3.3	Klassen	2-8
2.3.4	Assoziation	2-10
2.3.5	Aggregation und Komposition.....	2-11
2.3.6	Komponenten	2-13
2.3.7	Vererbung.....	2-14
2.3.8	Polymorphie.....	2-15
2.3.9	Sichtbarkeiten	2-16
2.3.10	Nachrichten.....	2-18
2.4	Die drei „K“ der Objektorientierung	2-19
2.4.1	Kapselung (encapsulation)	2-19
2.4.2	Kopplung (coupling)	2-24
2.4.3	Kohäsion (cohesion)	2-28
2.5	Zusammenfassung.....	2-30

2 Objektorientierte Programmierung

2.1 Einleitung

In diesem Kapitel werden wir uns mit einigen Grundsätzen und Begriffen beschäftigen. „Objektorientierte Programmierung“ setzt sich aus zwei Begriffen zusammen mit der Objektorientierung und der Programmierung. Folgerichtig werden wir im Folgenden beide Begriffe näher beleuchten.

Einleitung	 integrata cegos	
<p>Inhalt:</p> <ul style="list-style-type: none">▪ Grundsätze und Begriffe „Objektorientierte Programmierung“▪ Objektorientierung und Programmierung▪ Beide Begriffe werden hier näher beleuchtet▪ Vergleich Objektorientierung mit anderen Programmierer-Techniken▪ Konsequenzen für die folgenden Kapitel:<ul style="list-style-type: none">▪ zwei folgende Kapitel starke Schnittmenge zur Objektorientierten Analyse bzw. zum Objektorientierten Design.▪ Ohne OO Denkweise keine OO Programmierung▪ Diese Kapitel ist eine Wiederholung<ul style="list-style-type: none">▪ der OO Konzepte (gemeinsame Sprache und Verständnis)▪ UML als Basis zum Verständnis der Konzepte und Pattern im Design		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	2 Seite 2

Wir werden die Objektorientierung mit anderen Programmierer-Techniken vergleichen und daraus erste Konsequenzen für die folgenden Kapitel ableiten. Naturgemäß besitzen die folgenden zwei Kapitel eine starke Schnittmenge zur Objektorientierten Analyse bzw. zum Objektorientierten Design.

2.2 Arten der Programmierung

Lässt man spezielle und nur akademisch relevante Fälle außen vor, so können wir prinzipiell zwischen vier Programmier-Paradigmen unterscheiden.

Arten der Programmierung	 integrata cegos
<ul style="list-style-type: none">▪ Prozedurale Programmierung<ul style="list-style-type: none">▪ Cobol▪ C▪ Fortran▪ Funktionale Programmierung<ul style="list-style-type: none">▪ Haskell▪ LISP▪ Logische Programmierung<ul style="list-style-type: none">▪ Prolog▪ Objektorientierte Programmierung<ul style="list-style-type: none">▪ Java▪ C++▪ Smalltalk	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 3	

Prozedurale Programmierung

Prozedurale Programmiersprachen stellen die „klassische“ Art zu Programmieren dar. Ein Programm besteht aus einer Reihe von Anweisungen, die vom Rechner Schritt für Schritt, eine nach der anderen ausgeführt werden. Zwischenergebnisse können in Variablen abgelegt und mehrere Programmier-Schritte zu einem Unter-Programm (eben einer Prozedur) zusammengefasst werden.

Diese Arbeitsweise kommt dem Rechner sehr entgegen, da sie sehr dicht an der Funktionsweise eines Rechners selbst liegt. Damit wird der Aufwand, aus dem Quellcode ein ausführbares Programm herzustellen, deutlich geringer als bei den anderen Alternativen.

Typische Vertreter prozeduraler Programmiersprachen sind Basic, Cobol oder C.

Funktionale Programmierung

Funktionale Programmiersprachen, die auch als deklarativ¹ bezeichnet werden, beschreiben im Gegensatz zu prozeduralen Programmierspra-

¹ Wobei deklarative Sprachen eine Obermenge darstellt zu der auch die im Folgenden beschriebenen, logischen Sprachen gehören.

chen nicht, wie etwas zu berechnen ist, sondern stattdessen, was zu berechnen ist. Sie besitzen keinen inneren Zustand und keine Schleifen-Konstrukte, sondern ein Programm besteht aus einer Reihe von Funktionsdefinitionen, die mittels Komposition, Verzweigung und Rekursion zusammengesetzt werden. Ein Funktionsaufruf liefert dabei das Ergebnis zurück, hat aber sonst keine Seiteneffekte.

Funktionale Programmierung hat einige Zeit ein Schattendasein geführt und war schwerpunktmäßig an Universitäten interessant, erlebt aber in letzter Zeit eine Renaissance. Das ist nicht zuletzt in der Tatsache begründet, dass funktionale Programme viel leichter in eine Multi-Threading Umgebung umzusetzen sind, als das bei prozeduralen und objektorientierten Programmen der Fall ist. Mit der zunehmenden Anzahl an Mehrkern-Prozessoren wird diese Art der Programmierung dabei immer wichtiger.²

Typische Vertreter funktionaler Programmiersprachen sind Haskell, LISP und (als neue Sprache) Scala. Allen diesen Sprachen ist allerdings gemeinsam, dass sie Ansätze besitzen, die über einen reinen, funktionalen Kern hinausgehen.

Logische Programmierung

Logische Programmiersprachen folgen einem gänzlich anderen Ansatz. Ein Logik-Programm ist keine Sammlung von Anweisungen oder Funktionen, sondern besteht aus einer Menge von Fakten (Axiomen) und darauf aufbauenden Regeln. Eine Anfrage an das System ist dann die Frage nach einer Folgerung aus Fakten und Regeln, um weitere, abgeleitete Fakten zu generieren.

Das folgende, aus Wikipedia³ übernommene, einfache Beispiel soll das Prinzip verdeutlichen:

Fakten:

Lucia ist die Mutter von Minna.
Lucia ist die Mutter von Klaus.
Minna ist die Mutter von Nadine.

Regel:

Falls *X ist die Mutter von Y* **und** *Y ist die Mutter von Z* **Dann** *X ist die Großmutter von Z.*

Frage/Ziel:

Wer ist die Großmutter von Nadine?

Antwort des Computers, Folgerung aus den Fakten und Regeln:

Lucia

² Vergleiche hierzu auch das Kapitel über Nebenläufigkeit.

³ http://de.wikipedia.org/wiki/Logische_Programmierung

Wie aus dem Beispiel ersichtlich werden dürfte, ist die logische Programmierung nur für einen deutlich umrissenen Problembereich (vor allem künstliche Intelligenz bzw. Expertensysteme) sinnvoll.

Ein typischer Vertreter dieser Sprachen ist Prolog.

Objektorientierte Programmierung

Wie auch der prozeduralen liegt der objektorientierten Programmierung der Gedanke zu Grunde, dass dem Rechner Schritt für Schritt vorgegeben wird, was er tun soll. Der Unterschied liegt in der Verbindung von Anweisungen und Daten.

Ein Programm besteht in der objektorientierten Programmierung aus einem Zusammenschluss von einzelnen Objekten. Ein Objekt verbindet dabei eine Reihe von Daten (der Zustand) mit einem Verhalten (die Anweisungen, mit denen das Objekt gesteuert werden kann).

Die Vorteile der objektorientierten Programmierung liegen in der Tatsache, dass sich die Realität (bzw. ein System) objektorientiert in der Regel leichter formulieren lässt, als das mit ausschließlich prozeduralen Hilfsmittel möglich wäre. Außerdem erleichtert die objektorientierte Programmierung das Zerlegen eines Programms in einzelne Partitionen (Komponenten), was der Wiederverwendbarkeit zugute kommt.

Nachdem sich diese Unterlage konkret auf die objektorientierte Programmierung bezieht, werden wir uns mit diesem Thema natürlich im Folgenden noch ausführlicher auseinandersetzen.

Typische Vertreter objektorientierter Programmiersprachen sind C++, Java und Smalltalk.

Mischformen

In der Softwareentwicklung setzen sich seit einiger Zeit immer mehr Mischformen zwischen den einzelnen Arten zu Programmieren durch. So kann beispielsweise das steuernde Framework objektorientiert, die eigentliche Fachlichkeit aber funktional realisiert sein. Gerade im Java-Umfeld werden Kombinationen aus Java, Scala und einer Skriptsprache immer beliebter.

Ein Beispiel hierfür könnte eine Prozessmodellierung sein. Die Prozessablaufsteuerung selbst (der Server und/oder das Framework) sind in Java geschrieben, die einzelnen Business Komponenten in Java oder Scala und die (recht dynamischen) Entscheidungsknoten des konkreten Prozess dann in einer Skriptsprache (zum Beispiel Groovy oder JRuby).⁴


⁴ Skriptsprache bedeutet in diesem Fall, dass der Code nicht vorher kompiliert wird, sondern dynamisch erst zur Laufzeit des Programms.

2.3 Grundsätze objektorientierter Programmierung

Im folgenden Abschnitt wollen wir zunächst einige typische, objektorientierter Begriffe definieren, und uns dann mit den Prinzipien dieser Art zu Programmieren beschäftigen. Unser besonderes Ziel sollte dabei sein, die objektorientierte Programmierung dazu zu benutzen, die intrinsischen Qualitätsmerkmale unserer Software zu verbessern.

2.3.1 Unified Modeling Language (UML)

Wir werden die folgenden Punkte mit UML-Diagrammen ergänzen. UML ist eine formale, grafische Beschreibungssprache, die unter anderem dazu genutzt wird, Beziehungen zwischen Klassen zu visualisieren. Für alle wichtigen Konzepte gibt es in der UML festgeschriebene Notationen. Für unseren Anwendungsbereich reicht aber ein kleiner Ausschnitt der Sprache, den wir im Folgenden mit beleuchten wollen.

UML – Unified Modeling Language	
<ul style="list-style-type: none">▪ Kommt ursprünglich aus der OO-Softwareentwicklung▪ Die UML ist eine grafische Beschreibungssprache und anpassbar<ul style="list-style-type: none">▪ Die Anpassung erfolgt durch die Stereotypen▪ Aktuelle Version 2.x▪ Die UML besteht aus verschiedenen Diagrammen<ul style="list-style-type: none">▪ Diese basieren auf einem Meta-Modell▪ Die UML Diagramme können im Projekt in verschiedenen Phasen mit unterschiedlicher Intention eingesetzt werden▪ Hier nur lesend (um Konzepte zu verstehen)<ul style="list-style-type: none">▪ Klassen- und Paket-Diagramm reichen dafür aus▪ Konzeptionell um erste Strukturen beschreiben zu können<ul style="list-style-type: none">▪ Klassen- und Sequenz-Diagramm	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	2 Seite 4

2.3.2 Abstraktion

Jedes Objekt im System kann als ein abstraktes Modell eines *Akteurs* betrachtet werden, der Aufträge erledigen, seinen Zustand berichten und ändern und mit den anderen Objekten im System kommunizieren kann, ohne offenlegen zu müssen, wie diese Fähigkeiten implementiert sind. Solche Abstraktionen sind entweder Klassen (in der klassenbasierten Objektorientierung) oder Prototypen (in der prototypbasierten Programmierung).

Abstraktion		
<p>Jedes Objekt im System kann als ein abstraktes Modell eines <i>Akteurs</i> betrachtet werden, der Aufträge erledigen, seinen Zustand berichten und ändern und mit den anderen Objekten im System kommunizieren kann, ohne offenlegen zu müssen, wie diese Fähigkeiten implementiert sind.</p> <p>Solche Abstraktionen sind entweder Klassen (in der klassenbasierten Objektorientierung) oder Prototypen (in der prototypbasierten Programmierung).</p>		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	2 Seite 5

Grundsätzlich ist die Abstraktion kein Begriff, der sich ausschließlich auf die objektorientierte Programmierung bezieht, sondern auch durchaus in der prozeduralen Programmierung Anwendung findet. Er geht in der Objektorientierung allerdings noch weit über dieses Maß hinaus.

2.3.3 Klassen

In der Objektorientierung werden Objekte häufig durch ihre Klasse definiert. Eine Klasse fasst dabei gleichartige Objekte zusammen. Gleichartig bedeutet dabei, dass die Objekte ein vergleichbares Verhalten und die gleiche Art von Zustand besitzen. Eine Klasse beschreibt genau jenes Verhalten und die Art des Zustandes (mit eventuellen Standardwerten), die konkreten Objekte (die Instanzen der Klasse) den konkreten Zustand. Zustände bezeichnet man in der Objektorientierung als Attribute, Member- oder Instanzenvariablen, Fähigkeiten als Methoden

<h2>Klassen und Instanzen I</h2>	
<ul style="list-style-type: none">▪ Klassen<ul style="list-style-type: none">▪ Zusammenfassung gleichartiger Objekte (vergleichbares Verhalten und Zustand)▪ Beschreiben das Verhalten und die Art des Zustandes▪ Zustände heißen <i>Attribute</i>, <i>Member</i>- oder <i>Instanzvariablen</i>▪ Fähigkeiten/Verhalten heißen <i>Methoden</i>▪ Instanzen<ul style="list-style-type: none">▪ Konkrete Ausprägungen einer Klasse	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 6	

Beispiel:

Die Klasse „Kreis“ wird definiert durch den Mittelpunkt, den Radius sowie die Farbe. Jeder Kreis besitzt die Fähigkeit, seine Fläche und seinen Umfang zurück zu berechnen und zurück zu liefern.

Eine Instanz der Klasse Kreis besitzt den Mittelpunkt (10 3), den Radius 23 und die Farbe „Rot“. Eine weitere Instanz besitzt Mittelpunkt (0 0), den Radius 1 sowie die Farbe „Blau“.


Wir unterscheiden bei einer Klasse zwischen der äußeren und der inneren Sicht. Die äußere Sicht (die sogenannte Schnittstelle) beschreibt dabei die Fähigkeiten der Klasse, ohne konkret darauf einzugehen, wie diese realisiert werden. Die innere Sicht (der eigentliche Programmcode bzw. die Implementierung) beschreibt dagegen, wie die Methode tatsächlich funktioniert, also die einzelnen Anweisungen, die für diese Methode ausgeführt werden. Viele Programmiersprachen bieten die Möglichkeit, beide Sichten voneinander zu trennen⁵.

Eine Klasse wird in UML als Kasten dargestellt, der in drei Bereiche eingeteilt ist. Zuerst steht der Name der Klasse, gefolgt von den Attributen und zuletzt den Methoden (bzw. in der UML-Terminologie: Operationen). Interfaces, also die Trennung von Schnittstelle und Implementierung (und damit auch vom Zustand) werden als „Klasse“ darge-

⁵ Z.B. bei Java durch Interfaces, bei C++ durch Header-Dateien.

stellt, die keinen Attribut-Bereich besitzt und zusätzlich mit dem Stereotyp «interface» markiert ist.⁶

Klassen und Instanzen II



- Äußere Sicht (Schnittstelle)
- Innere Sicht (Implementierung)

Kreis
x : float y : float radius : float Farbe : String
berechneUmfang() : float berechneFläche() : float

«interface» Kreis
berechneUmfang() : float berechneFläche() : float

- Merksatz:

{Instanz} ist ein (konkreter) {Klasse}

Hans ist ein konkreter Mensch

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 7

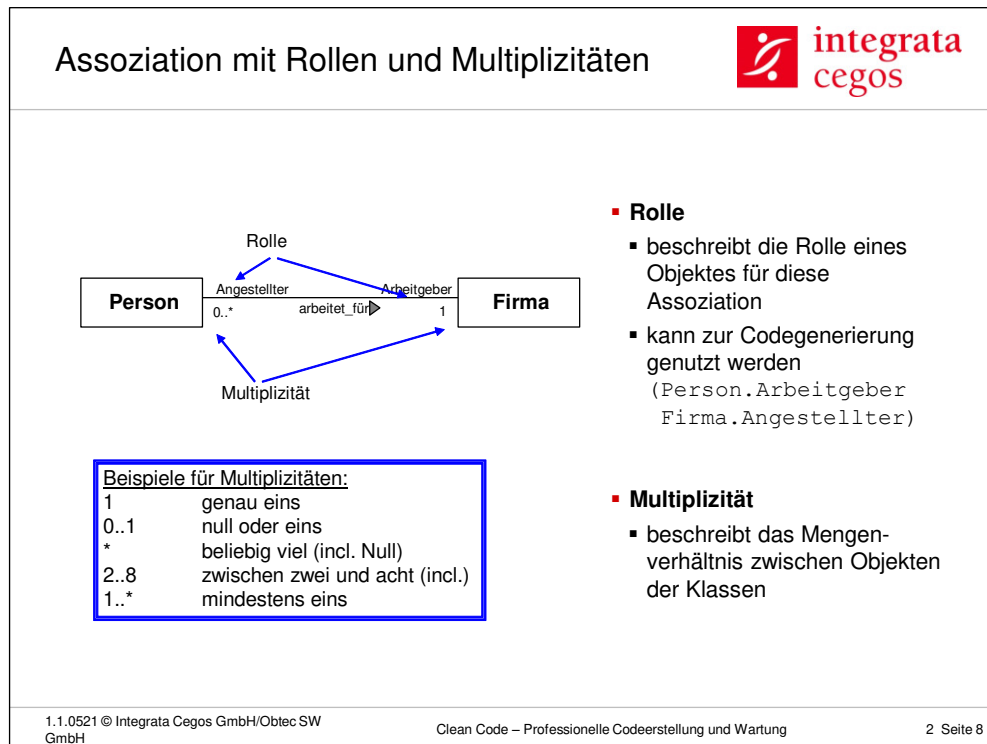
2.3.4 Assoziation

Eine Assoziation stellt eine deutlich leichtere Verbindung von einem Objekt zu einem anderen (bzw. einer Klasse zu einer anderen) dar. Eine Assoziation wird formuliert als „kennt“. Kennen bedeutet in diesem Fall, die Fähigkeiten des anderen Objektes kennen und damit auch darauf zugreifen können. Eine andere Sichtweise wäre: Der Name der bekannten Klasse taucht im Code der kennenden Klasse auf und ist auch zur Kompilierzeit der Klasse verfügbar (ggf. als Header-Datei oder Interface).

Kompositionen sind in der Regel auch Assoziationen, für Aggregationen gilt das häufig auch (aber nicht so häufig wie bei Kompositionen). Ein Auto, das unter anderem aus einem Motor besteht, wird normalerweise auch die Klasse Motor kennen, umgekehrt wird der Motor häufig auch Zugriff auf sein Auto haben und es damit kennen. Denkbar ist allerdings auch, dass dieses Kennen explizit nur in Gegenrichtung der Komposition / Aggregation existiert.

⁶ Ein Stereotyp ist eine Möglichkeit, in UML zusätzliche Informationen unterzubringen, die an anderer Stelle definiert wurden. Stereotypen werden in französischen Anführungszeichen eingefasst («stereotyp»). Einige Stereotypen (wie «interface») sind in der UML-Definition vorgelegt.

In UML werden Assoziationen als einfache Linien dargestellt, seit UML immer mit Pfeilspitzen, die die Richtung angibt. Ist diese nicht vorhanden, so bedeutet das, dass sie nicht bekannt ist.



2.3.5 Aggregation und Komposition

Objekte, deren Zustände ihrerseits wieder durch Objekte beschrieben werden, sind sogenannte Kompositionen. Abgegrenzt ist dieser Begriff zur Aggregation, die im Programmcode in der Regel identisch aussieht (auch hier sind die Attribute eines Objektes wieder Objekte), aber eine inhaltlich etwas unterschiedliche Bedeutung hat. Eine Komposition bedeutet dabei eine so enge Verknüpfung, dass die Bestandteile (die Komponenten) in der Regel alleine keinen Sinn machen bzw. ihre Lebensdauer an die des Komposites (des „Besitzers“) geknüpft sind.

Komposition und Aggregation I



- Im Code in der Regel identisch (Attribute)
- Aggregation
 - Strengere Assoziation
 - "Enthält" Beziehung
 - Oder „Besteht aus“ Beziehung
- „Personengruppe besteht aus Personen“
- „Kontoverwaltung enthält Konten“
- Komposition:
 - Wie Aggregation nur
 - Das Komposit ist alleine nicht „lebensfähig“
 - Ein Auto **besteht aus** vier Reifen, einem Motor und einer Karosserie.
 - Ein Motor **ist Bestandteil** eines Autos.
 - In anderem Kontext auch als Aggregation möglich

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code – Professionelle Codeerstellung und Wartung

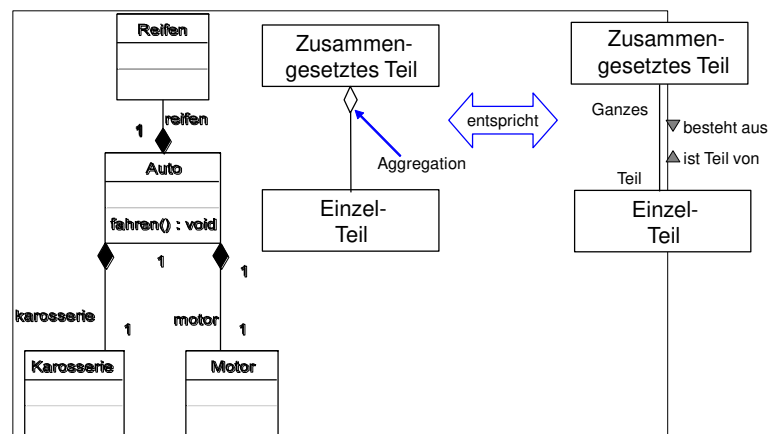
2 Seite 9

Eine gängige Sprechweise für Komposition ist „besteht aus“ bzw. in umgekehrter Richtung „ist Bestandteil von“.

Komposition und Aggregation II



- Ein Objekt kann immer nur Teil einer einzigen Komposition sein.
- Eine Komposition ist immer gerichtet.



1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code – Professionelle Codeerstellung und Wartung

2 Seite 10

Grundsätzlich ist die Frage, ob eine Beziehung zwischen zwei Objekten eine Aggregation oder eine Komposition darstellt, auch immer durch die Fachlichkeit beeinflusst. Trotzdem wollen wir zumindest zwei Grundsätze beschreiben, die bei der Auswahl nützlich sein könnten:

- Ein Objekt kann immer nur Teil einer einzigen Komposition sein. Taucht das Objekt als Attribut weiterer Objekte auf, sind diese Beziehungen zwangsläufig Aggregationen.
- Eine Komposition ist immer gerichtet. Ist eine Beziehung beidseitig, so kann nur eine Richtung die Komposition sein, die andere muss zwangsläufig eine Aggregation sein (*Ein Auto besteht (unter anderem) aus einem Motor, ein Motor hat eine direkte Beziehung zu seinem Auto*).

Im UML-Diagramm werden Kompositionen als Linie mit einer ausgefüllten Raute auf Besitzer-Seite dargestellt, bei Aggregationen ist die Raute nicht gefüllt. Die beiden Seiten der Linien werden mit Angaben der Kardinalität versehen. Diese wird in üblicher Form mit 1, 1..*, 0..* oder einer festen Zahl dargestellt. Gegenüber der Kardinalität kann die Rolle der Zielklasse angegeben werden, die diese in der Assoziation belegt, üblicherweise steht hier der Name des Attributes, unter dem die Zielklasse im Besitzer abgelegt ist. Ist kein Name angegeben, so wird der Name der Zielklasse angenommen. Bei einer Komposition ist die Kardinalität auf Komposite-Seite logischerweise immer 1.

2.3.6 Komponenten


Eine Komponente ist eine weitere Abstraktionsebene oberhalb der Klasse. Es handelt sich um den Zusammenschluss einer Reihe von Klassen zu einem gemeinsamen Zweck. Dabei werden weitere Implementierungsdetails verborgen und die Komponente zeigt nach außen eine oder mehrere Schnittstellen. Die Bestandteile einer Komponente sind dabei in der Regel über Komposition und Aggregation mit einander verknüpft. Eine Komponente kann wiederum aus Unterkomponenten bestehen. Auf diese Weise entsteht eine Architektur, die bei jedem Schritt einen Abstraktionslevel tiefer geht. Je nach Programmiersprache gibt es Sprachelemente, die entweder direkt Komponenten darstellen oder dazu genutzt werden können.⁷

Komponenten sind wieder über Assoziationen mit anderen Komponenten verknüpft.

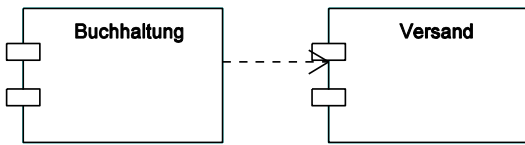
Eine Komponente wird in UML als Kasten mit zwei kleinen Rechtecken auf der linken Seite dargestellt.⁸

⁷ In Java sind das beispielsweise Pakete (packages)

⁸ In UML 2 wird die Komponente als Rechtecke mit dem «component» Classifier und einem Komponentensymbol in der rechten oberen Ecke dargestellt. Für unsere Zwecke ist aber auch diese Unterscheidung unerheblich.

Komponenten


- Weitere Abstraktionsebene oberhalb der Klasse
- Besitzen auch innere und äußere Sicht (Klassen)
- Können geschachtelt sein
- In Java über Pakete (packages) realisierbar



1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 11

2.3.7 Vererbung

Klassen können in der Regel voneinander erben. Man bezeichnet das Erben auch als Spezialisierung einer Basisklasse. Die erbende Klasse übernimmt damit alle Fähigkeiten der Oberklasse, kann diese aber gegebenenfalls erweitern oder überschreiben. Auch der Zustand wird geerbt, wobei nicht gesagt ist, dass die Unterklasse auch direkten Zugriff auf ihren ererbten Zustand hat (vgl. Sichtbarkeiten). Der Zweck der Vererbung liegt darin, Gemeinsamkeiten zusammen zu fassen, was insbesondere im Zusammenspiel mit der Polymorphie (vgl. folgenden Abschnitt) sehr saubere und zukunftsichere Architekturen ermöglicht.

Man kann eine Vererbungsbeziehung auch als „ist ein“ (oder „ist eine besondere Form von“) Satz formulieren:

*Ein Angestellter **ist eine** (spezielle) Person*

*Ein Hund **ist ein** Tier.*

Aus äußerer Sicht (der Schnittstellen-Sicht) stellt eine Unterklasse immer nur eine Erweiterung ihrer Oberklasse dar, d.h. eine Fähigkeit der Oberklasse kann einer Unterklasse nicht wieder aberkannt werden. Aus innerer Sicht kann sich die Unterklasse aber sehr wohl anders verhalten.

Erbt eine Klasse von einem Interface (also auf Implementierungsebene), so nennt sich die Vererbung auch „Realisierung“ bzw. Implementierung.

Vererbung

- Klassen *erben* voneinander
- Eine Unterklasse ist eine *Spezialisierung* der Oberklasse
- Bei Interfaces: Realisierung oder Implementierung
- Die Oberklasse wird dadurch erweitert oder verändert
- Gemeinsamkeiten werden zusammengefasst
- Sparen von *Schreibarbeit* ist nicht das Ziel
- *A ist ein (spezieller) B*
 - Ein Angestellter **ist eine** (spezielle) Person
 - Ein Hund **ist ein** Tier.
- *Fähigkeiten der Oberklasse können nicht „aberkannt“ werden!* (aus Schnittstellen-Sicht)

```

classDiagram
    class Auto {
        fahren() void
    }
    class Benzinauto {
        tanken() void
    }
    class Angestellter {
        <<interface>>
    }
    class Zeitarbeiter
    Auto <|-- Benzinauto
    Angestellter <|.. Zeitarbeiter
    
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 12

In UML wird eine Spezialisierung durch einen Pfeil mit einer dreieckigen Spitze dargestellt (die zur Basisklasse zeigt), bei der Realisierung ist der Pfeil selbst gestrichelt dargestellt.

2.3.8 Polymorphie

Ein wichtiges Konzept der Vererbung ist die Polymorphie (Vielgestaltigkeit). Sie sagt aus, dass ein Objekt auch als eine Instanz seiner Oberklasse angesprochen werden kann, bzw. dass der aufrufende Code nicht einmal Kenntnis darüber benötigt, um was für eine Klasse es sich bei dem agierenden Objekt konkret handelt.

Ein Beispiel:


Die Klasse Auto besitzt eine Methode „fahren“. Damit ist garantiert, dass jede Unterklasse von Auto diese Fähigkeit auch besitzt (nicht aber, was die jeweilige Unterklasse im Einzelnen tut). Fügen wir nun dem Beispiel zwei Unterklassen hinzu: Elektroauto und Benzinauto. Beide können „fahren“, allerdings mit unterschiedlichen Ergebnissen – so wird der Verbrauch sicher unterschiedlich berechnet und von anderen Betriebsstoffen abgezogen.

Eine Klasse Person, die nur das Auto kennt (aber nicht die Unterklassen), kann trotzdem alle drei Auto-Arten fahren, da sie nur Fähigkeiten nutzen muss, die ihr auch bekannt sind.

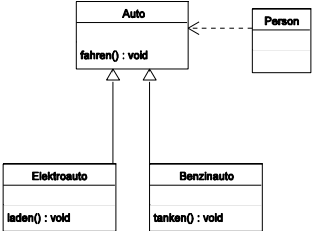
Erst zur Laufzeit wird, anhand der tatsächlichen Klasse des Autos, entschieden, welche Methode tatsächlich aufgerufen wird.

Die Person ist allerdings nicht in der Lage, die Autos aufzutanken, da die Fähigkeit „tanken“ bzw. „laden“ Teil der jeweiligen Unterklasse ist.

Polymorphie



- Eine Instanz kann auch als Instanz der Oberklasse angesprochen werden (Schnittstelle)
- Welche Methode (tatsächlicher Code) ausgeführt wird zur Laufzeit anhand der tatsächlichen Klasse entschieden
 - Der Compiler prüft anhand des „Etiketts“ (Variablentyp), ob die Methode aufgerufen werden *darf*
 - Das System entscheidet zur Laufzeit, *welcher* Code ausgeführt wird
- Ein Aufrufender Code muss die konkrete Unterklasse, deren Methode ausgeführt wird, nicht kennen! (i.S.d. Assoziation)




```

classDiagram
    class Auto {
        <fahren() : void
    }
    class Elektroauto {
        <laden() : void
    }
    class Benzinauto {
        <tanken() : void
    }
    class Person
    Auto <|-- Elektroauto
    Auto <|-- Benzinauto
    Auto ..> Person
      
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 13

2.3.9 Sichtbarkeiten

Sichtbarkeiten definieren, wer welche Methode und Felder einer Klasse nutzen kann. Welche Sichtbarkeiten konkret zur Verfügung stehen, ist von der genutzten Programmiersprache abhängig. Die vier Sichtbarkeiten, die die Sprache Java bietet, decken aber den Normalfall ab und finden sich auch ähnlich in der UML-Darstellung wieder. Zum Vergleich werden wir hierbei auch die C++-Sichtbarkeiten aufführen. In UML werden Sichtbarkeiten durch ein einzelnes Symbol vor dem Feld oder der Methode dargestellt (was natürlich in der Implementierungssicht wirklich Sinn macht).

Sichtbarkeiten	
<ul style="list-style-type: none">▪ Gelten für Klassen, Methoden und Felder▪ Public (+)<ul style="list-style-type: none">▪ Jeder darf die Methode aufrufen▪ Private (-)<ul style="list-style-type: none">▪ Die Methode darf nur von innerhalb der gleichen Klasse aufgerufen werden▪ Protected (#)<ul style="list-style-type: none">▪ Nur die Klasse und ihre Unterklassen dürfen die Methode aufrufen▪ Unter Java: Eigentlich eine Kombination aus Package und Instance-Protected▪ Package (~)<ul style="list-style-type: none">▪ Die Methode ist nur für Klassen im gleichen Package (Komponente) erreichbar▪ Gibt es in C++ nicht▪ Instance-Private / Instance-Protected<ul style="list-style-type: none">▪ Wie Private und Protected, nur darf der Aufruf nur aus der selben <i>Instanz</i> erfolgen▪ Nicht Teil der UML-Sichtbarkeiten▪ Gibt es z.B. in Ruby und Scala	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 14	

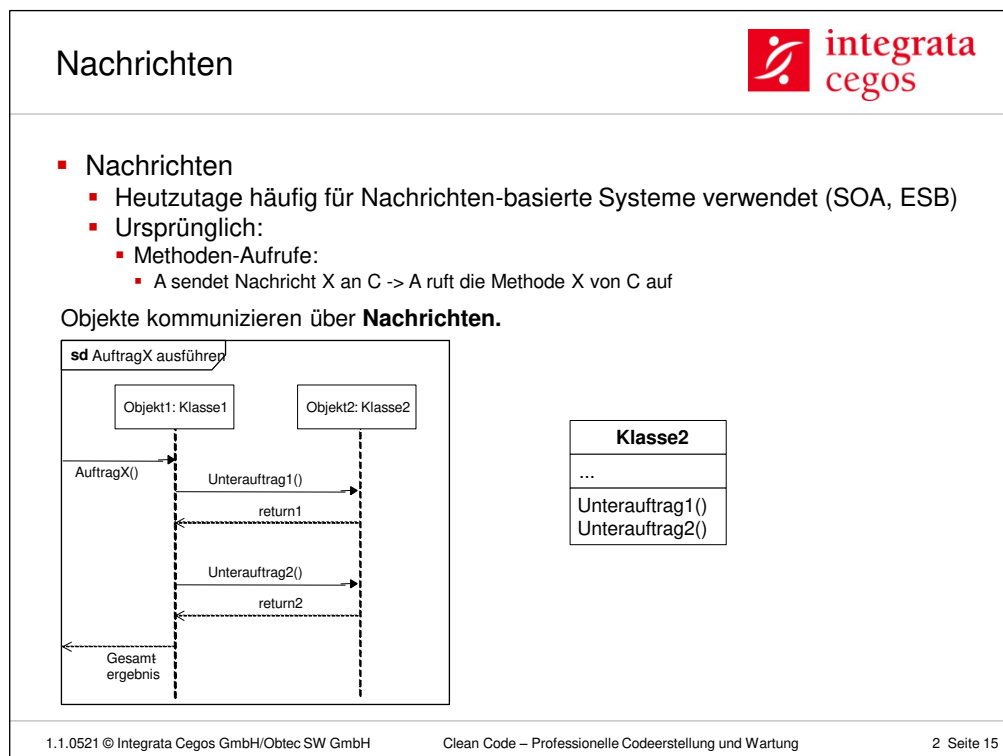
- **Public(+):** *Public* Methoden oder Felder sind für alle sicht- bzw. nutzbar, unabhängig davon, aus welcher Klasse Sie aufgerufen werden. Das gilt sowohl für C++ als auch für Java.
- **Private(-):** *Private* Methoden und Felder sind nur in der Klasse sichtbar, in der sie definiert wurden – also auch nicht in Unterklassen. Auch das ist in C++ und Java gleich.
- **Protected(#):** *Protected* Methoden und Felder sind unter C++ nur für die Klasse und ihre Unterklassen sichtbar. Darüber hinaus sind *protected* Methoden unter Java zusätzlich auch für Klassen im gleichen Paket sichtbar.
- **Package(~):** Die Methoden und Felder sind nur für Klassen im eigenen Paket sichtbar. C++ hat keine direkte Entsprechung für diese Sichtbarkeit, unter Java wird sie auch als *default*-Sichtbarkeit bezeichnet, da sie gültig ist, wenn nicht explizit eine Sichtbarkeit angegeben ist.

In der Praxis gehören Sichtbarkeiten zu den Elementen, bei denen es sich selten lohnt, sie im UML-Diagramm aufzuführen, da sie sich sowieso im Laufe der Zeit ändern werden. Ein einfacher Ansatz lautet: Attribute sind standardmäßig *private* (dazu später mehr) und werden nur dann markiert, wenn es Gründe gibt, davon abzuweichen. In das Diagramm eingetragene Methoden sind dagegen standardmäßig *public*. Grundsätzlich bleibt dann zu klären, in wieweit nicht-*public* Methoden überhaupt in das Diagramm aufgenommen werden sollten.

2.3.10 Nachrichten

Auch der Nachrichtenbegriff ist mittlerweile mehrfach belegt. Heute versteht man darunter in der Regel eine Nachricht (also Objekt oder Datenstruktur), die mittels eines Dienstes innerhalb eines Programms oder zwischen Programmen und Rechnern, meist asynchron, ausgetauscht wird. Für diese Art der Programmierung, die in erster Linie der Entkopplung von Teilsystemen gilt, gibt es einige „Buzz-Words“, wie zum Beispiel „Message-oriented-Middleware (MOM)“ oder „Enterprise Service Bus (ESB)“. Auch das Schlagwort „SOA (Service oriented architecture)“ zielt oftmals in diese Richtung.

In der ursprünglichen Sprachweise der Objektorientierung bedeutet „eine Nachricht versenden“ aber lediglich, eine Methode eines Zielobjektes aufzurufen bzw. auszuführen. Mit dieser Terminologie sollte der Eigenständigkeits-Gedanke der Objekte unterstrichen werden.




2.4 Die drei „K“ der Objektorientierung

Nachdem wir uns bisher mit den (handwerklichen) Grundsätzen der Objektorientierung beschäftigt haben, werden wir uns als nächstes mit den Regeln für gutes Design auseinandersetzen. Es gibt in der Objektorientierung drei Eigenschaften, die gutes Design fördern können. Diese bezeichnen wir im Folgenden als „die drei K der Objektorientierung“.

2.4.1 Kapselung (encapsulation)

Die Kapselung (auch: „Datenkapselung“) hat ihren Ursprung in den Abstrakten Datentypen in der prozeduralen Programmierung. Grundgedanke dabei ist, dass der Zugriff auf die eigentlichen Daten nur über definierte Prozeduren möglich ist. Damit wird die Datenstruktur nicht mehr durch ihren internen Aufbau, sondern durch ihr Verhalten bestimmt⁹.

Die drei „K“ der Objektorientierung – Kapselung	 integrata cegos	
<ul style="list-style-type: none">▪ Klassen erlauben Zugriff auf ihre Internas nur über wohldefinierte Schnittstellen▪ Implementierungsdetails werden verborgen▪ „Versehentlicher“ Zugriff wird verhindert▪ Implementierungsdetails können geändert werden, ohne dass der nutzende Code neu kompiliert werden muss▪ Anderer Name: Geheimnis-Prinzip▪ Werkzeuge:<ul style="list-style-type: none">▪ Sichtbarkeiten<ul style="list-style-type: none">▪ Unterscheidung zwischen öffentlicher und protected Schnittstelle▪ Schnittstellen sollten stabil gehalten werden▪ Zugriffsmethoden (Getter und Setter)<ul style="list-style-type: none">▪ Dienen auch der Dokumentation (das Attribut wird Teil der öffentlichen Schnittstelle)▪ Erlaubt Überprüfung von Invarianten und Einschränkungen▪ Schaffung von <i>virtuellen</i> Attributen		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	2 Seite 16

In der objektorientierten Programmierung bedeutet Kapselung, dass Implementierungsdetails verborgen werden und ein Objekt nur über eine wohldefinierte Schnittstelle genutzt oder verändert werden kann. Typische Implementierungsdetails sind dabei Felder (Instanzvariablen) und Hilfsmethoden.

Indem mit technischen Hilfsmitteln verhindert wird, dass der Nutzer eines Objektes direkt auf Implementierungsdetails zugreift, hat der Programmierer der Klasse des Objektes die Möglichkeit, diese Details je-

⁹ Was eine Vorstufe zur echten Objektorientierung darstellt.

derzeit zu verändern, ohne die Funktionalität andere Elemente zu gefährden¹⁰.

Werkzeuge zur Kapselung sind die oben erwähnten Sichtbarkeiten und Zugriffsmethoden (sog. Getter und Setter). Bei letzterem handelt es sich um Methoden, die in der Regel nur ein Feld auslesen und zurückliefern oder das Feld neu setzen.

2.4.1.1 Sichtbarkeiten

Durch eingeschränkte Sichtbarkeiten verhindern wir, dass bestimmte Methoden von außen aufgerufen werden. Als Faustregel gilt: alles was zur öffentlichen Schnittstelle gehört, sollte auch als *public* deklariert sein. Was nicht dazu gehört, sollte eine eingeschränkte Sichtbarkeit besitzen (welche, werden wir gleich noch beleuchten).

Mit jeder öffentlichen Methode definieren wir einen Vertrag (englisch: Contract) über das Verhalten unserer Klasse. Ändern wir eine öffentliche Methode in ihrem Verhalten (ob nun in der Signatur oder im Algorithmus), so verändern wir auch diesen Vertrag, mit dem Ergebnis, dass wir sicherstellen müssen, dass jeder Code, der diese Methode nutzt, ebenfalls geändert (oder zumindest überprüft) werden muss. Jede öffentliche Methode stellt also einen potentiellen Kopplungspunkt für andere Klassen dar. Dass das einen großen Aufwand nach sich ziehen kann, dürfte klar sein. Um diesen klein zu halten, definieren wir eine Regel:

Regel 2-1: Die Signatur und das Verhalten von Schnittstellen-Methoden sollte nachträglich nur noch in Ausnahmefällen geändert werden (Wart, Wied).

Das hat umgekehrt die Konsequenz, dass wir die Schnittstellen eben so klein wie möglich halten sollten.

Regel 2-2: Neue Methoden sollten der Schnittstelle nur dann hinzugefügt werden, wenn es dafür einen konkreten Anwendungsfall gibt (Wart).

Damit bleibt nur die Frage offen, welche Sichtbarkeit statt *public* vergeben werden soll. Ein sinnvoller Ansatz ist es, so eingeschränkt wie möglich zu programmieren, d.h. grundsätzlich ist jedes Feld und jede Methode, die nicht zur öffentlichen Schnittstelle gehört, zunächst *private* und wird erst bei einer Notwendigkeit sichtbar gemacht.

Der Vorteil der Lösung über *private* ist, dass auf diese Art auch eine *innere Kapselung* erreicht wird. Dabei kapselt sich eine Klasse nicht nur gegen fremde Klassen, sondern effektiv auch gegen „freundliche“ Klassen (sprich Klassen im selben Paket / in derselben Komponente) und ihre eigenen Unterklassen ab.

¹⁰ Weil die Implementierung dabei „geheim“ bleibt, ist ein weiterer Name „Geheimnisprinzip“

Ob das sinnvoll und notwendig ist, hängt von den Umständen ab. So erleichtert die innere Kapselung natürlich auch interne Veränderungen an der Klasse. Das wird aber durch zusätzlichen Aufwand erkauft. Gibt man statt dessen Feldern und internen Methoden die Sichtbarkeit *package*, so entsteht zwangsläufig eine engere Kopplung zwischen den Klassen des Pakets. In der Regel ist der damit verbundene Zusatzaufwand aber vertretbar, es sei denn, die Komponente ist ungewöhnlich groß, stark volatil oder wird durch einen weit verzweigten Entwicklerkreis weiterentwickelt.

Ob die Sichtbarkeit *protected* Sinn macht, hängt vor allem davon ab, ob die Klasse dazu entwickelt wurde, durch andere Entwickler bzw. in anderen Projekten abgeleitet zu werden (also Teil einer offenen API ist). Ist das der Fall, so schafft man effektiv eine zweite Schnittstelle mit der Sichtbarkeit *protected*, für die die beiden oben aufgeführten Regeln ebenso gelten müssen.

Es gibt aber zumindest einen guten Grund für den Verzicht von *private* zugunsten von *package*: Unit-Tests. Wir werden uns mit dem Thema Testen später noch näher befassen, das wichtigste soll aber hier schon vorab genannt sein: Unit-Test sind separate Klassen (Testklassen), deren Aufgabe es ist, die Funktionalität ihrer zu testenden Klasse sicherzustellen. Sie können entweder als reine Schnittstellen-Tests (sog. *Black-Box-Tests*) implementiert werden, bei denen ausschließlich die öffentliche Schnittstelle getestet wird, oder aber als *White-Box-Tests*, die eben auch die Interna (also beispielsweise Hilfsmethoden) testen. Zu bevorzugen ist natürlich eine Kombination aus beidem, wobei die *White-Box-Tests* natürlich eng mit dem eigentlichen Code gekoppelt sind und deshalb bei Änderung i.d.R. mit angefasst werden müssen.

Aber was hat das mit der Sichtbarkeit zu tun? Nun, wenn *White-Box-Tests* Internas testen wollen, so müssen sie auch Zugriff auf diese haben, was mit *private* nicht gestattet wäre. Liegen die Tests aber im gleichen Paket wie die zu testende Klasse, so reicht die Sichtbarkeit „*package*“ aus, um diesen Zugriff zuzulassen.¹¹


Fassen wir noch einmal in einer Regel zusammen:

Regel 2-3: Felder sollten standardmäßig *private*, Hilfsmethoden standardmäßig *package-visible* sein. (Wart, Test)

¹¹ Existiert in der Sprache keine Paket-Sichtbarkeit (wie beispielsweise in C++), wo muss man adäquate Alternative suchen (im Falle von C++ z.B. *friendly*)

2.4.1.2 Getter und Setter

Effektiv haben wir damit also den direkten Feldzugriff im Code durch den Zugriff auf eine Methode ersetzt. Das scheint in erster Linie noch keinen großen Unterschied zu machen, aber der Schein trügt.

Hinweise zu Gettern und Settern	
<ul style="list-style-type: none">▪ Einige Sprachen lassen den Zugriff auf Attribute grundsätzlich nur über Getter und Setter zu (z.B. Smalltalk)▪ Je nach Sprache können Zugriffsmethoden eine Performance-Einbuße mit sich bringen<ul style="list-style-type: none">▪ aber nicht in Java▪ Es ist zu entscheiden, über die Kapselung auch gegenüber den eigenen Unterklassen erfolgen sollte (private vs. protected Attribute)	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 17	

Dadurch, dass die Zugriffsmethoden eben Methoden sind, sind sie Teil der Schnittstelle unserer Klasse. Das hat einige positive Konsequenzen:

- Durch die Aufnahme in die Schnittstelle machen wir deutlich, dass wir erwarten, dass dieses Attribut von anderen genutzt wird. Damit haben wir auch eine gute Stelle, um Konsequenzen des Zugriffs zu dokumentieren (Seiteneffekte).
- Dadurch, dass der Zugriff über eine Methode läuft, können wir den Zugriff kontrollieren. Damit können zum Beispiel Einschränkungen auf das Feld (nur positive Werte) oder Invarianten auf die ganze Klasse durchgesetzt werden.

Gleichzeitig wäre die „Invariante“ damit in diesem Fall eigentlich gar keine Invariante, sondern lediglich eine Regel. Mit einer Setter-Methode kann aber schon beim Setzen des Wertes festgestellt werden, ob dieser gegen Regeln verstößt und damit eine echte Invariante umgesetzt werden. Außerdem muss die Überprüfung so nur einmal stattfinden.

Bleibt noch zu erwähnen, dass das die Verwendung von Zugriffsmethoden für den aufrufenden Code nur minimale Veränderungen erfordert.

Bei einigen Programmiersprachen ist die Benutzung von Zugriffsmethoden sogar gar nicht zu vermeiden. So sind beispielsweise in Smalltalk Felder immer *private*, d.h. der Zugriff **kann** nur über Getter und Setter erfolgen.

- Die Getter und Setter müssen nicht zwangsläufig auf real existierende Felder verweisen. Sie können auch sogenannte *virtuelle Felder* beschreiben, also Felder, bei denen die Schnittstelle nur so aussieht, als ob diese Felder existieren. Das ist nützlich, wenn der Wert eines Feldes sich aus einem anderen berechnen lassen kann (Radius und Durchmesser eines Kreises), oder aber, wenn die Implementierung – also konkret die Felder – einer Klasse im Nachhinein geändert werden müssen.

Wird nun später in der Entwicklung entschieden, dass der innere Zustand nicht mehr aus *top*, *left*, *width* und *height* bestehen soll, sondern sinnvollerer zwei gegenüberliegende Ecken mit ihren Koordinaten angegeben werden sollen (also *top*, *left*, *bottom*, *right*), so muss jeder Code, der die Klasse benutzt im Zuge dieser Veränderung angepasst werden – was einen immensen Aufwand nach sich ziehen würde.

Hätte man stattdessen die Klasse mit Gettern und Settern modelliert, so hätte die Umstellung keinerlei Auswirkungen auf aufrufenden Code, Auf diese Art und Weise haben wir natürlich die Wartungsfreundlichkeit unsere Klasse deutlich verbessert.

Getter und Setter zu verwenden, hat natürlich auch Nachteile:

- Der Code selbst wird deutlich länger, und das in der Regel um Inhalte, die keine Informationen bringen (sog. Boilerplate-Code).
- Je nach Anwendung und Programmiersprache kann das Verwenden einer zusätzlichen Methode Geschwindigkeitseinbußen mit sich bringen.


Zum ersten Nachteil gibt es je nach Programmiersprache einige Ansätze, um das Problem zu reduzieren, z.B. in dem die Zugriffsmethoden automatisch im Buildprozess generiert werden.

Der zweite Nachteil ist je nach Programmiersprache minimal bzw. gar nicht vorhanden (in neueren Java-Versionen ist die Zugriffszeit sogar identisch!). Grundsätzlich sollte man aber vermeiden, aus Geschwindigkeitsgründen vom guten Stil abzuweichen (es sei denn, es liegt ein **konkretes** Performanceproblem vor, vgl. hierzu das Kapitel Optimierung).

2.4.2 Kopplung (coupling)

Kopplung beschreibt, wie eng zwei Klassen zusammenhängen, bzw. wie eigenständig die Klassen (bzw. Komponenten¹²) sind. Umgekehrt sind sie daher ein Maß dafür, wie viele Änderungen das Ändern einer abhängigen Komponente nach sich zieht.

Je loser die Kopplung, d.h. je weniger Abhängigkeiten zu anderen Klassen existieren, desto isolierter ist eine Klasse. Das erhöht die Wiederverwendbarkeit und – wie wir später noch sehen werden – auch die Testbarkeit einer Klasse.

Kopplung I		
<ul style="list-style-type: none">▪ Beschreibt, wie eng zwei Klassen zusammenhängen▪ Wie stark wirkt sich eine Änderung der einen Klasse auf die andere aus (muss diese auch angepasst werden?)▪ Inhaltsskopplung<ul style="list-style-type: none">▪ Eine Klasse greift auf Internas der anderen Klasse zu▪ Eine Änderung der einen wird wahrscheinlich auch eine notwendige Änderung der anderen Klasse nach sich ziehen.▪ Schnittstellenkopplung<ul style="list-style-type: none">▪ Klassen sind nur über ihre Schnittstelle an einander gekoppelt▪ Die Klassen „kennen“ die konkreten Implementierungen nicht▪ Implementierungen können verändert oder sogar ausgetauscht werden, ohne die aufrufende Klasse zu beeinflussen (solange der Vertrag der Methoden eingehalten wird)		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	2 Seite 21

Wir unterscheiden zwischen verschiedenen Arten der Kopplung:

2.4.2.1 Inhaltsskopplung (enge Kopplung)

Die Klasse greift auf Internas der jeweils anderen Klasse zu. Das wichtigste innere Detail sind dabei natürlich die Felder, die eben nicht direkt zugreifbar sein sollten. Inhaltsskopplung ist damit ein Gegenstück zur Kapselung. Anders formuliert: eine starke Kapselung vermeidet eine Inhaltsskopplung.


Eine Inhaltsskopplung stellt eine enge Kopplung dar, die wir unbedingt vermeiden wollen. Sind zwei Klassen inhaltlich eng gekoppelt, so wird

¹² Kopplungen zwischen Klassen und Kopplungen zwischen Komponenten folgen exakt den gleichen Prinzipien, nur in unterschiedlichen Größenordnungen. Das gleich gilt auch für die später angesprochene Kohäsion. Wir werden im Folgenden der Lesbarkeit halber nur noch von Klassen sprechen und Komponenten implizit einschließen.

jede Änderung der einen Klasse eine potentielle Änderung der anderen Klasse nach sich ziehen, was die Wartbarkeit natürlich immens verschlechtert. Auch die Verständlichkeit leidet deutlich unter einer engen Kopplung, da man zum Verständnis einer Klasse zwangsläufig auch die andere verstehen muss.

2.4.2.2 Schnittstellenkopplung (lose Kopplung)

Gibt es ein derartiges Konzept in der verwendeten Programmiersprache, so sind Interfaces das beste Mittel, um enge Kopplungen zu vermeiden. Da der aufrufende Code hier in der Regel nur die Schnittstelle zu sehen bekommt, kann er eben auch nicht auf Implementierungsdetails zugreifen. Gibt es kein eigenes Interface-Konzept in der Sprache, so werden stattdessen abstrakte Klassen ohne konkrete Implementierung verwendet.

Kopplung II	 integrata cegos	
<ul style="list-style-type: none">■ Probleme der Schnittstellen-Kopplung<ul style="list-style-type: none">■ An einer Stelle muss konkret eine Instanz der zu benutzenden Klasse erzeugt werden■ Damit „kennt“ die aufrufende Klasse die zu nutzende doch■ Mögliche Lösungen:<ul style="list-style-type: none">■ FACTORY-Pattern■ DEPENDENCY INJECTION-Pattern (INVERSION OF CONTROL)■ Einige Sprachen bieten implizite Unterstützung dafür an■ Datenkopplung<ul style="list-style-type: none">■ Fähigkeiten werden nicht mehr über Methoden beschrieben, sondern über eine allgemeine (meist Text-basierte) Sprache■ Die Ausführende Klasse hat nur noch eine Methode „ausführen“, was ausgeführt werden soll, wird dieser Methode als Parameter übergeben■ Minimalste Kopplung, erlaubt Sprachen-übergreifende Aufrufe (Bsp.: Webservices)■ Nachteile:<ul style="list-style-type: none">■ Keine Compiler-Überprüfung■ Schwer Lesbar		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	2 Seite 22

Das entbindet den Programmierer natürlich nicht von der Notwendigkeit, seine Klasse stark zu kapseln, er sollte die Klasse so programmieren, als würde ein Client-Code direkt (ohne das Interface) auf die Klasse zugreifen. Anders formuliert: die Klasse selbst sollte nicht mehr Implementierungsdetails preisgeben, als es das Interface tut.

Regel 2-4: Jede Klasse sollte mit einem entsprechenden Interface gekapselt sein. Client-Code sollte ausschließlich über das Interface auf die Klasse zugreifen. (Wart, Wied, Test)

Im Idealfall sollte also die konkrete Implementierung im Quellcode der aufrufenden Klasse gar nicht vorkommen. Anders formuliert: im UML-Diagramm sollte keine Assoziation von einer Klasse zu einer anderen Klasse, sondern immer nur zu einem Interface erfolgen.

Eine weitere Möglichkeit, die die Schnittstellenkopplung uns bietet, ist die Implementierung nicht nur zu verändern, sondern vollständig auszutauschen.

Ein einfaches Beispiel soll das verdeutlichen:

Damit das Prinzip allerdings vollständig umgesetzt werden kann, darf auch die konkrete Instanz des Interfaces niemals direkt in der aufrufenden Klasse erzeugt werden, sonst wäre ja zumindest an dieser einen Stelle eine enge Kopplung gegeben. Um diese Dilemma zu lösen kann man sich eines von zwei Design Patterns (dazu später mehr) zunutze machen:

- **Das FACTORY-Pattern:** Nicht die Klasse selbst erzeugt das Objekt, sondern eine Hilfsklasse, deren einziger Zweck es ist, Instanzen zu erzeugen, die das Interface implementieren (welche konkreten Instanzen erzeugt werden, muss natürlich irgendwo festgelegt werden. Damit hat die Factory zwar potentiell eine enge(re) Kopplung zu den Implementierungen, aber eben nicht mehr die aufrufende Klasse.
- **Das DEPENDENCY INJECTION Pattern (auch bekannt als INVERSION OF CONTROL oder IoC):** Die Klasse holt sich ihre Abhängigkeiten überhaupt nicht selbst, sondern bekommt diese von außen „untergeschoben“, durch Setter-Methoden oder im Konstruktor.

Natürlich sollte abschließend die Frage erlaubt sein, wann denn von dieser Regel abgewichen werden darf oder sollte. Grundsätzlich ist dieser Ansatz umso wichtiger, je volatiler (veränderlicher) die Zielklasse ist. Ist diese in einem stabilen Zustand (wie das zum Beispiel für Basis-klassen und reine Datenklasse häufig der Fall ist), so ist das Erstellen eines zusätzlichen Interfaces nicht notwendig und führt ggf. sogar zu unnötiger Komplexität.¹³

¹³ Niemand würde unter Java für die Klasse String ein extra Interface verwenden – selbst wenn dies möglich wäre.

2.4.2.3 Datenkopplung (freie Kopplung)

Datenkopplung bedeutet, dass die eigentliche Kopplung nicht mehr über eine fachliche Schnittstelle erfolgt, sondern nur noch über ein gemeinsames Austauschformat. Das könnte zum Beispiel ein String sein, der die auszuführende Anweisung enthält. (Statt dass eine Methode `berechneSteuer()` aufgerufen wird, wird eine Methode `tuEs()` mit dem Parameter `berechneSteuer` aufgerufen). Es handelt sich damit effektiv um den Austausch des Protokolls, mit dem die beiden Klassen mit einander sprechen, von einem sprachabhängigen Protokoll (dem Aufruf von Methoden) zu einem selbst-implementierten Modell.

Wo liegt nun der Vorteil dieser Methode? Wir erreichen dadurch die kleinst-mögliche Kopplung zwischen den Komponenten (auf Klassenebene macht die Datenkopplung wenig Sinn). Eine Komponente kann damit zur Laufzeit Funktionalität aufrufen, die zum Zeitpunkt der Programmierung noch überhaupt nicht bekannt war.


Außerdem wird damit erleichtert, mit anderen Programmiersprachen zu kommunizieren, da es nun nicht mehr notwendig ist, dass alle Sprachen die gleichen Fähigkeiten besitzen. Es reicht im obigen Beispiel schon aus, dass die Zielsprache mit Zeichenketten umgehen kann – sie muss nicht einmal objektorientiert sein. Diese freie Kopplung ist das Prinzip der Service-Oriented-Architectures (SOA), die beispielsweise über WebServices mit einander kommunizieren.

Der Nachteil dieser Kopplung ist allerdings, dass hier der Compiler keine Hilfestellungen mehr liefern kann (z.B. bei Tipp-Fehlern). Außerdem leidet die Lesbarkeit des Codes in der Regel deutlich.

Deshalb sollte freie Kopplung nur dann eingesetzt werden, wenn es fachlich wirklich notwendig ist.

2.4.3 Kohäsion (cohesion)

Kohäsion bedeutet wörtlich übersetzt Zusammenhalt oder Bindung. Es beschreibt die fachliche/inhaltliche Bindung der Elemente einer Klasse untereinander. Aus fachlicher Sicht besitzt eine Klasse also dann eine hohe Kohäsion, wenn ihre Felder und Methoden fachlich zusammengehören. Ein erster Prüfstein hierfür ist der Klassenname. Passt dieser zu allen Methodennamen bzw. zu allen Feldern, verfügt die Klasse vermutlich über eine hohe Kohäsion¹⁴.

Kohäsion


- Wörtlich: Zusammenhalt, Bindung
- Methoden und Attribute müssen „zusammenpassen“
- Indiz für schwache Kohäsion
 - Bestimmte Attribute werden nur von bestimmten Methoden benutzt
 - Partitionen (Cluster) von Attribut/Methoden-Gruppen

```

public class Warteschlange {
    private int naechstesElement = 0; // Feld (1)
    private List<Person> eintraege
        = new LinkedList<Person>(); // Feld (2)

    public int groesse() { // nutzt (1)
        return naechstesElement;
    }

    public void hinzufuegen(Person p) { // nutzt (1) und (2)
        naechstesElement++;
        eintraege.add(p);
    }

    public Person naechstePerson() { // nutzt (1) und (2)
        if (naechstesElement == 0)
            throw new SchlangeLeerException();
        naechstesElement--;
        Person result = eintraege.get(naechstesElement);
        eintraege.remove(naechstesElement);
        return result;
    }

    public String personAlsString(Person p) {
        return p.getVorname() + " " + p.getNachname();
    }
}

```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code – Professionelle Codeerstellung und Wartung
2 Seite 24

Aus technischer Sicht ist ein gutes Maß für Kohäsion, wie viele Felder eine Methode benutzt. Je mehr Felder sie nutzt, desto enger hängt die Methode mit ihrer Klasse zusammen. Eine maximal kohäsive Klasse wäre demnach eine Klasse, bei der sämtliche Methoden sämtliche Felder nutzen. Dass das in der Realität kaum zweckmäßig wäre, dürfte klar sein (zumal damit ja auch Getter und Setter nicht benutzt werden dürften). Trotzdem sollte unsere Kohäsion möglichst hoch sein. Folgt eine Klasse nur einem bestimmten Zweck, so ist ihre Wiederverwendbarkeit deutlich höher, als wenn sie mehrere Verantwortlichkeiten gleichzeitig bedient.


¹⁴ „Passen“ bedeutet hier aus der Sicht eines Sprechers der benutzen Sprache (Deutsch, Englisch, ...), nicht der Programmiersprache!

Die ersten drei Methoden besitzen eine starke Kohäsion, die letzte Methode allerdings überhaupt keine. Das ist ein guter Hinweis darauf, dass diese Methode hier fehl am Platze ist und besser ausgelagert werden sollte (z.B. in die Klasse `Person`).

Regel 2-5: Klassen sollten eine starke Kohäsion besitzen (Wied, Lesb, Vers)

2.5 Zusammenfassung

Wir haben uns in diesem Kapitel mit den Grundsätzen der objektorientierten Programmierung auseinandergesetzt. Insbesondere haben wir wesentliche Begriffe wiederholt und dabei die drei „K“ der Objektorientierung näher beleuchtet.

Die drei „K“ – Ziele	 integrata cegos
<ul style="list-style-type: none">▪ Gute Kapselung▪ Lose Kopplung▪ Hohe Kohäsion ▪ Wir haben erste Regeln für professionellen Code aufgestellt – auf diese wollen wir im nächsten Kapitel weiter aufbauen.	
<div>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code – Professionelle Codeerstellung und Wartung2 Seite 26</div>	

Wir haben erste Regeln für professionellen Code aufgestellt – auf diese wollen wir im nächsten Kapitel weiter aufbauen.

3

Professionelle Klassen und Objekte

3.1	Einleitung.....	3-3
3.2	Klassenhierarchien	3-4
3.2.1	Das Liskovsche Substitutionsprinzip (LSP).....	3-6
3.2.2	Die Holper-Regel	3-7
3.2.3	Mehrfachvererbungen.....	3-8
3.3	Schnittstellen	3-10
3.3.1	Hierarchieschnittstellen.....	3-10
3.3.2	Fähigkeitsschnittstellen.....	3-12
3.3.3	Mix-Ins	3-15
3.3.4	Parallele Schnittstellen-Hierarchien	3-16
3.3.5	Client-Schnittstellen	3-18
3.3.6	Das Interface-Segregation-Principle ISP	3-18
3.4	Klassengrößen	3-22
3.4.1	Das Visions-Prinzip.....	3-24
3.4.2	Das Single-Responsibility-Principle – SRP	3-25
3.5	Änderungen ermöglichen	3-27
3.5.1	Das Open-Closed-Principle – OCP).....	3-27
3.5.2	Das Dependency-Inversion-Principle – DIP).....	3-29
3.6	Zusammenfassung	3-31

3 Professionelle Klassen und Objekte

3.1 Einleitung

In diesem Kapitel bauen wir auf den Grundlagen des vorherigen Kapitels auf, und beschäftigen uns mit den Details, die eine gute Klasse von einer professionellen Klasse unterscheiden. Wir beschäftigen uns insbesondere mit dem Zusammenspiel mehrerer Klassen, sei es über Vererbung oder Assoziationen.

Weiterhin legen wir einige Regeln und Prinzipien fest, die wir auch in weiteren Kapiteln immer wieder aufgreifen werden.

Insbesondere gehen wir dabei auf fünf Prinzipien zur Klassenmodellierung ein, die Robert C. Martin in einem Artikel¹ formuliert bzw. zusammengefasst² hat. Dazu kommen allerdings weitere Ansätze, wie beispielsweise KISS.

Professionelle Klassen – Einleitung	 integrata cegos	
<ul style="list-style-type: none"> ▪ Aufbau auf Grundlagen des vorherigen Kapitels <ul style="list-style-type: none"> ▪ Details, die eine gute Klasse von einer professionellen Klasse unterscheiden ▪ Insbesondere mit dem Zusammenspiel mehrerer Klassen ▪ sei es über Vererbung oder Assoziationen. ▪ Regeln und Prinzipien <ul style="list-style-type: none"> ▪ Fünf Prinzipien zur Klassenmodellierung <ul style="list-style-type: none"> ▪ Robert C. Martin SOLID ▪ Nicht alle diese Prinzipien stammen von ihm selbst. ▪ Weitere Regeln und Prinzipien <ul style="list-style-type: none"> ▪ KISS ▪ Design Pattern 		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	3 Seite 2


¹ <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>

² Nicht alle diese Prinzipien stammen von ihm selbst.

3.2 Klassenhierarchien

Zunächst wollen wir uns mit Klassenhierarchien auseinandersetzen und dabei insbesondere mit der Frage, wann es sinnvoll ist, eine Klasse von einer anderen abzuleiten. Um diese Frage zu motivieren, beginnen wir mit einem kleinen Beispiel:

Es gibt eine Klasse Person und eine Klasse Buchhaltung. Beide besitzen eine Methode `darstellungsName()`, der den fachlichen Namen der jeweiligen Instanz in einer nutzerfreundlichen Form zurückliefert.

Klassenhierarchien


- Was wäre eine passende, gemeinsame Oberklasse?

Person
<code>darstellungsName() : String</code>

Buchhaltung
<code>darstellungsName() : String</code>

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code – Professionelle Codeerstellung und Wartung
3 Seite 3

Eine Möglichkeit, die sich leicht aufzudrängen scheint, ist es, eine gemeinsame Oberklasse zu schaffen, die eben die Methode `darstellungsName()` definiert (wahrscheinlich abstrakt). Aufdrängen deshalb, weil leider die erste Begründung für Vererbung die Vermeidung von doppeltem Code zu sein scheint. Tatsächlich wird diese Begründung in Ausbildung und Unterricht aber nur deshalb oft zuerst genannt, weil diese Begründung deutlich leichter zu verstehen und einzusehen ist, als das Konzept der Polymorphie.

Tatsächlich ist es unzumutbar bzw. in der Regel sogar schlichtweg falsch, hier eine gemeinsame Oberklasse einzubringen. Der erste und vornehmlichste Zweck der Vererbung ist eben nicht, Code einzusparen, sondern Beziehungen zwischen den beteiligten Klassen darzulegen (ist-ein-Beziehungen, bzw. Spezialisierungen). Bringt man eine Klasse in eine Vererbungsbeziehung mit einer anderen Klasse, mit der Sie in-

haltlich nichts zu tun hat, so missbraucht man den Mechanismus, wo-
runter zu allererst die Verständlichkeit leidet.

Natürlich gibt es Situationen, in denen Beziehungen notwendigerweise
gegen diesen Grundsatz verstoßen (allen voran haben objektorientierte
Sprachen ja in der Regel eine gemeinsame Oberklasse, von der alle
Klassen ableiten – im Falle von Java ist das `java.lang.Object`), und
wir werden weiter unten auch einen Mechanismus kennenlernen, mit
dem wir die ja ohne Zweifel vorhandenen Gemeinsamkeiten zusam-
menfassen können, ohne die Vererbung zu verbiegen.

Regel 3-1: Vererbung sollte nur dazu benutzt werden, tatsächliche Spezialisierungen zu beschreiben. (Vers)

Allerdings ist hier zumindest in bestimmten Fällen Vorsicht geboten,
manchmal kann die Spezialisierungseigenschaft auch in die Irre führen.


Ein klassisches Beispiel ist das Kreis-Ellipse-Problem:

	<h3>Das Kreis-Ellipse Problem</h3> <ul style="list-style-type: none"> ▪ <i>Es gibt eine Klasse Kreis und eine Klasse Ellipse. Aus geometrischer Sicht ist folgende Aussage sicher richtig: Ein Kreis ist eine spezielle Ellipse (bei der eben beide Halbachsen gleich lang sind).</i> ▪ Problem: <ul style="list-style-type: none"> ▪ Methoden <code>skaliereX()</code> und <code>skaliereY()</code> können in der Unterklasse (Kreis) nicht adäquat umgesetzt werden ▪ Polymorphie funktioniert damit möglicherweise nicht wie erwartet.
<div style="display: flex; justify-content: space-between; font-size: small;"> 1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH Clean Code – Professionelle Codeerstellung und Wartung 3 Seite 5 </div>	

Leider kann das aus objektorientierter Sicht je nach Anwendungsfall in die falsche Richtung führen. Unsere Ellipse könnte beispielsweise zwei Methoden `skaliereX()` und `skaliereY()` besitzen (beispielsweise in einem Grafikprogramm). Da Kreis eine Unterklasse von Ellipse ist, würde Kreis diese beiden Methoden erben, die aber für einen Kreis unzulässig wären, da nach einer Anwendung der Kreis wahrscheinlich kein Kreis mehr wäre.

3.2.1 Das Liskovsche Substitutionsprinzip (LSP)

Eine Möglichkeit, dieses Problem zu erkennen, bietet das Liskovsche Substitutionsprinzip³. Es lautet folgendermaßen:

Das Liskovsche Substitutionsprinzip (LSP)	 integrata cegos	
<i>„Sei $q(x)$ eine beweisbare Eigenschaft von Objekten x des Typs T. Dann soll $q(y)$ für Objekte y des Typs S wahr sein, wobei S ein Untertyp von T ist.“</i>		
Übersetzung:		
<i>Überall, wo die Oberklasse verwendet wird, muss auch bedenkenlos eine Instanz der Unterklasse eingesetzt werden können.</i>		
oder:		
<i>Alle Tests, die auf eine Instanz der Oberklasse korrekt ausgeführt werden, müssen auch auf Instanzen der Unterklasse korrekt ausgeführt werden können.</i>		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	3 Seite 6

Versuchen wir, diesen ziemlich akademischen Satz auf unser Kreis-Ellipse-Problem anwenden: Unsere beweisbare Eigenschaft lautet in diesem Fall *„die Achsen können unabhängig voneinander skaliert werden“* (beweisbar bedeutet in diesem Fall: die Methoden werden korrekt ausgeführt und das Ergebnis ist das zu erwartende⁴).

Versuchen wir noch zwei weitere Formulierungen:

*Überall, wo die Oberklasse verwendet wird, muss auch bedenkenlos eine Instanz der Unterklasse eingesetzt werden können.*⁵

Oder, im Vorgriff auf Unit-Tests:

Alle Tests, die auf eine Instanz der Oberklasse korrekt ausgeführt werden, müssen auch auf Instanzen der Unterklasse korrekt ausgeführt werden können.

³ 1993 von Barbara Liskov und Jeannette Wing formuliert

⁴ Ansonsten wäre es ja auch denkbar, dass `skaliereX()` und `skaliereY()` für einen Kreis einfach das gleiche tun würden. Das wäre aber keinesfalls das zu erwartende Ergebnis.

⁵ Was ja das Grundprinzip der Polymorphie ist.

Das Liskovsche Substitutionsprinzip ist eines der erwähnten fünf Prinzipien von Robert C. Martin. Allerdings hat er sich einer einfacheren Formulierung bedient, die ungefähr unserer ersten Umdeutung entspricht:

Regel 3-2: (LSP) Unterklassen müssen an die Stelle ihrer Oberklassen treten können. (Test, Vers)

3.2.2 Die Holper-Regel

Offen bleibt natürlich die Frage, wie man erkennt, ob eine Vererbung sinnvoll ist oder nicht. Im vorangegangenen Kapitel haben wir die Holper-Regel definiert, die wir im Folgenden noch um eine zweite Regel erweitern wollen:

Regel 3-3: (Holper-Regel) Lässt sich für die Anwendung einer objektorientierten Technik kein vernünftiger (nicht-holpriger) Name finden, so ist die Technik nicht korrekt angewendet. (Vers)

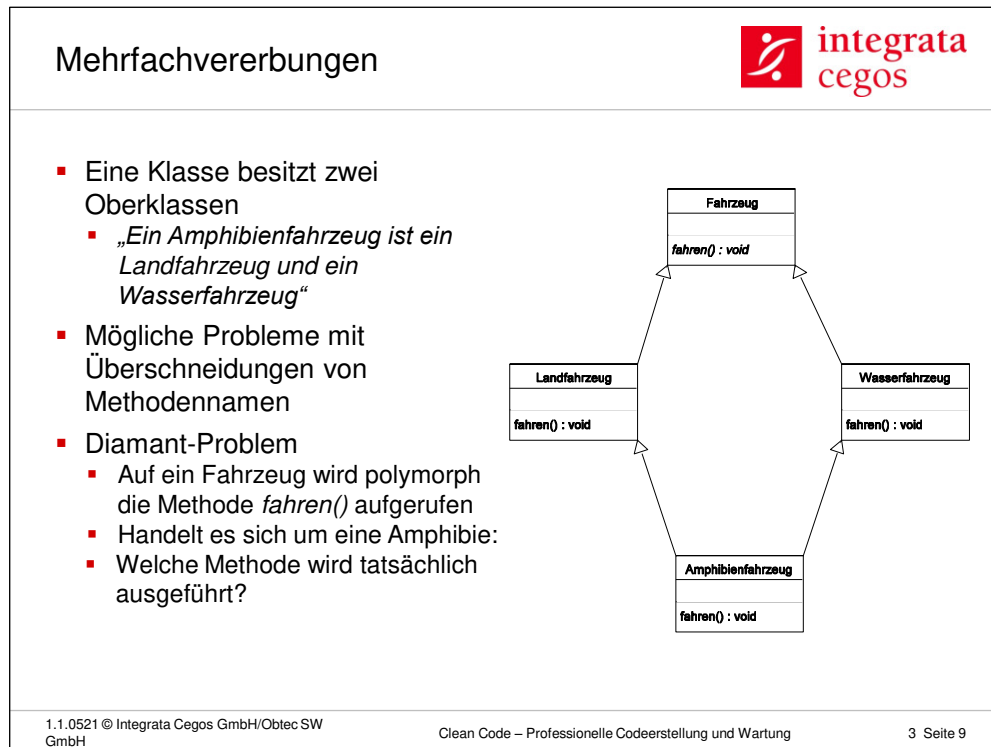
Wenden wir die Holper-Regeln auf das obige Beispiel an. Wie könnte eine gemeinsame Oberklasse heißen? Zwei wenig sinnvolle Möglichkeiten wären `DarstellungsObjekt` oder `DarstellbaresObjekt`. Beides klingt schon ziemlich gekünstelt. Auch die Sprechweise klingt nicht besser:

Eine Person ist eine (besondere Form) eines darstellbaren Objektes.

Beide Holper-Regeln deuten also deutlich darauf hin, dass eine gemeinsame Oberklasse hier nicht das Mittel der Wahl ist. Bleibt natürlich die Frage: Wenn nicht so, wie dann? Schließlich wollen wir ja trotzdem doppelten Code vermeiden.

3.2.3 Mehrfachvererbungen

Eine mögliche Lösung wäre die Verwendung von Mehrfachvererbungen. Hierbei handelt es sich um eine Architektur, in der eine Klasse von zwei Oberklassen erbt. In der Sprechweise wäre das eine „und“ Verknüpfung von zwei „ist-ein“-Bedingungen.



Ein Amphibienfahrzeug ist ein Landfahrzeug und ein Wasserfahrzeug.

Zunächst erscheint der Satz vollständig sinnvoll und logisch, verstößt also nicht gegen die Holper-Regel. Allerdings gibt es im Detail einige Probleme:

- Es kann zu Überschneidungen bei Methoden und Attributnamen kommen, die über sprachliche Mittel behandelt werden müssen, was die Lesbarkeit in der Regel deutlich erschwert.
- Es kann zum klassischen Diamant-Problem kommen, bei der eine Klasse von zwei Oberklassen erbt, die wieder von einer gemeinsamen Klasse erben.

In unserem Beispiel erben sowohl Landfahrzeug als auch Wasserfahrzeug von Fahrzeug. Besitzt nun Fahrzeug eine Methode `fahren()`, welche Methode wird dann aufgerufen, wenn auf einer Instanz eines Amphibienfahrzeuges `fahren()` aufgerufen wird? Die des Landfahrzeuges, des Wasserfahrzeuges oder beide? Wird das Fahrzeug konkret als Wasserfahrzeug oder als Landfahrzeug angesprochen, ist der Fall recht klar. Bei einem Amphibienfahrzeug direkt

kennt der Entwickler zumindest die Problematik und kann mit angeben, welche Methode gemeint ist.

Problematisch wird es aber, wenn die Methode `fahren()` auf ein beliebiges Fahrzeug aufgerufen wird, das tatsächlich ein Amphibienfahrzeug ist. Möglicherweise kennt der aufrufende Code die Unterklassen gar nicht und hat somit gar keine Chance, auszuwählen, welche Methode ausgeführt wird.

Relativ schnell fällt auf, dass diese Struktur natürlich das Liskovsche Substitutionsprinzip zumindest gefährdet. Um dieses Problem zu behandeln, haben unterschiedlich Programmiersprachen verschiedene Lösungen bereitgestellt. Die einfachste lautet: Mehrfachvererbung ist nicht erlaubt, zumindest nicht was die Implementierung angeht (Java erlaubt beispielsweise, dass eine Klasse mehrere Interfaces (also Schnittstellen*beschreibungen*) implementiert, aber konkrete Methoden werden nur von einer einzigen Klasse, nämlich der Oberklasse geerbt, womit das obige Problem effektiv verhindert wird).

In der Theorie führen Mehrfachvererbungen häufig zu Lagerkämpfen, in der Praxis zu schwer nachvollziehbarem Code, was uns zur nächsten Regel motiviert:

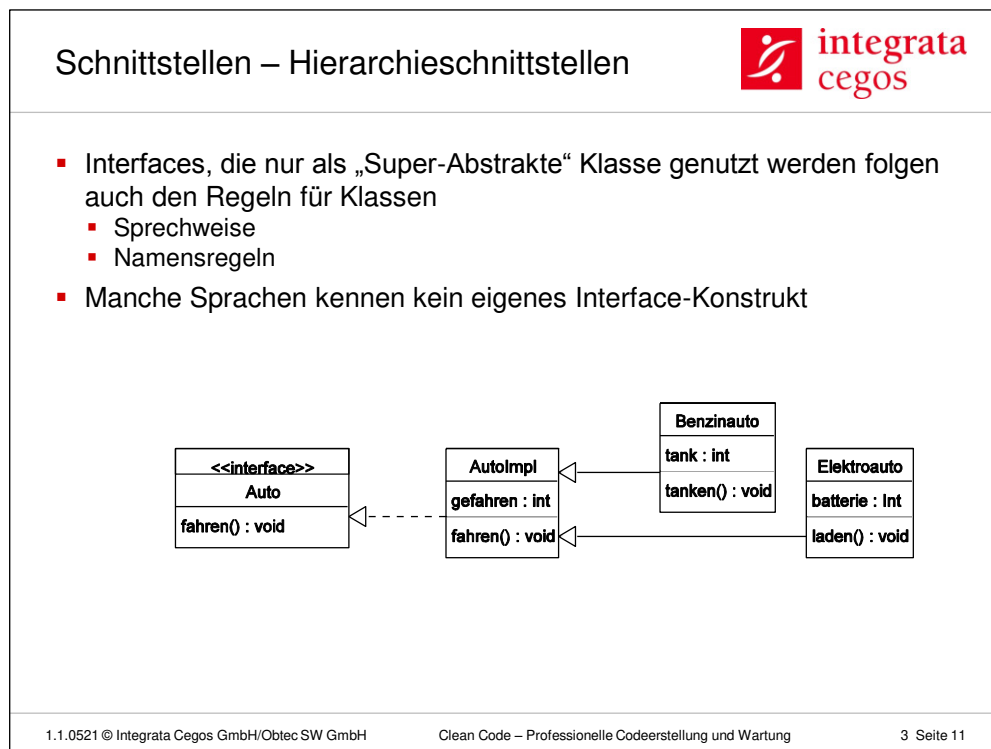
Regel 3-4: Mehrfachvererbung ist zu vermeiden. (Lesb, Vers)

3.3 Schnittstellen

3.3.1 Hierarchieschnittstellen

Interfaces haben wir bisher als Möglichkeit kennengelernt, Schnittstellen von der Implementierung zu trennen. In diesem Fall erfüllt die Schnittstelle normalerweise die Aufgabe einer abstrakten Klasse in der Vererbungshierarchie, d.h. die Schnittstelle selbst wird wie eine Oberklasse behandelt⁶. Diese Schnittstellen wollen wir, wenn die Unterscheidung deutlich gemacht werden soll, als *Hierarchieschnittstelle* bezeichnen. Sie ist aus objektorientierter Sicht keine echte Schnittstelle, sondern eigentlich eine abstrakte Klasse ohne implementierte Methoden und ohne Felder⁷.

Für Hierarchieinterfaces gelten im Großen und Ganzen die gleichen Regeln wie für normale Klassen auch: Der Name sollte ein Substantiv sein, auf hohe Kohäsion und gute Kapselung ist zu achten (wobei gerade die Kapselung hierbei natürlich nur deutlich schwächer ausgeprägt ist – gute Kapselung bedeutet hierbei, dass nur Dinge in die Schnittstelle aufgenommen werden, die auch wirklich öffentlich gemacht werden sollen).



⁶ Für Sprachen, die kein eigenes Sprachelement für Schnittstellen besitzen, ist das sowieso der Normalfall.

⁷ Dennoch wird man in Sprachen, die es unterstützen, das Sprachelement für Schnittstellen dafür verwenden (z.B. Interfaces in Java)

Die Sprechweise ist die gleiche, wie bei der normalen Vererbung, und damit gilt auch die Holper-Regel. Eine alternative Formulierung, die für die direkte Beziehung zwischen `AutoImpl` und `Auto` einen saubereren Satz ergibt, lautet „ist eine Implementierung von“ oder „ist ein konkretes“.

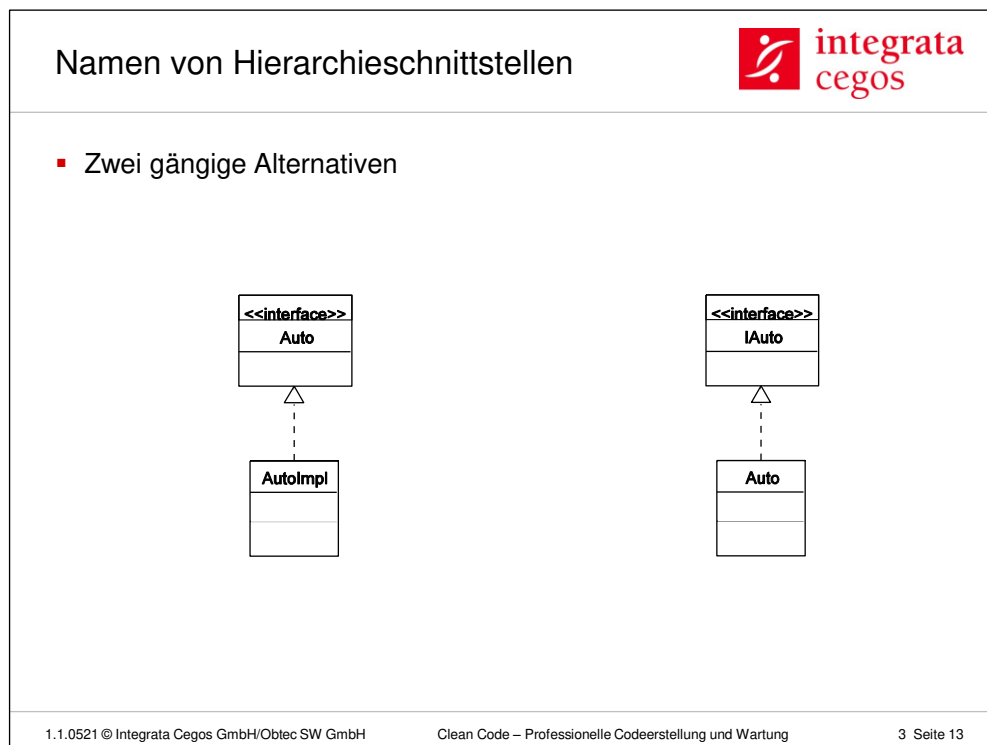
Weil eine Hierarchieschnittstelle prinzipiell eine (maximal) abstrakte Klasse ist und wir weiter oben bereits Mehrfachvererbungen ausgeschlossen haben, können wir hier unsere Regel noch ein wenig verdeutlichen:

Regel 3-5: Eine Klasse sollte immer nur entweder von einer Oberklasse ableiten *oder* eine Hierarchieschnittstelle implementieren. (Vers)

Umgekehrt wird eine Hierarchieschnittstelle häufig nur eine direkte Implementierung besitzen (wenn man von Dummy-Implementierungen o.ä. für Tests absieht).

Namen von Hierarchieschnittstellen

Wir sollten uns einen Moment mit der Vergabe von Namen bei Hierarchieschnittstellen beschäftigen. Zwischen einer Schnittstelle und ihrer direkt implementierenden Klasse existiert zwangsläufig eine sehr enge Bindung, die sich auch im Namen wiederfindet. Hier existieren zwei gängige Varianten:



Der Autor bevorzugt die erste Variante. Wenn wir uns die Regeln der Kopplung zu Eigen machen, dann werden wir möglichst immer nur mit den Schnittstellen und nicht mit den konkreten Implementierungen arbeiten, der Name der Schnittstelle wird also häufig in unserem Code vorkommen (je nach Variante also `Auto` oder `IAuto`). Da das zusätzliche „I“ keine neue Information einbringt, den Lesefluss aber möglicherweise hemmt, erscheint die erste Variante die sinnvollere. Das können wir auch wieder als Regel definieren:

Regel 3-6: Der Name des Konstruktes (Klasse, Schnittstelle), der im Code am häufigsten verwendet wird, sollte der „schönste“ sein. (Lesb)

Mit der Namensvergabe werden wir uns in einem späteren Kapitel noch ausführlicher beschäftigen.

Bleibt hinzuzufügen, dass in beiden Sprachvarianten abstrakte Klassen, die nur dazu dienen, gemeinsame Funktionalität vorzudefinieren (also ein reines, **technisches** Hilfsmittel für Ableiter sind) mit dem Wort „Abstract“ oder „Abstrakt“ beginnen sollten (also zum Beispiel `AbstractAuto`).

3.3.2 Fähigkeitsschnittstellen

Offen ist aber immer noch die Frage, wie wir unser Darstellungsproblem aus Abschnitt 0 lösen. Versuchen wir es zunächst mal mit einem besseren Satz (kehren wir also die Holperregel um):


*Eine **Person** ist **darstellbar**.*

*Eine **Buchhaltung** ist **darstellbar**.*

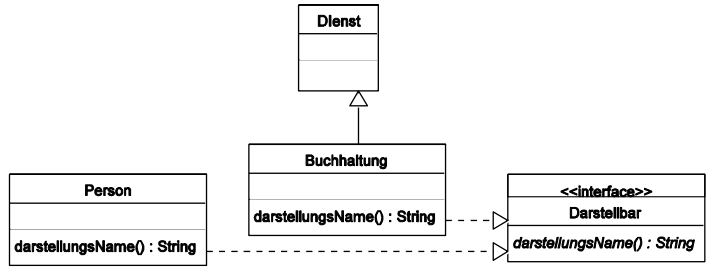
Klingt zumindest nicht mehr holprig. Allerdings haben wir jetzt statt eines zweiten Substantivs ein Adjektiv in unserem Satz. Hierfür bietet die Objektorientierung ein weiteres Konzept an, das der Fähigkeitsschnittstelle. Eine Fähigkeitsschnittstelle beschreibt, was ein Objekt einer Klasse tun kann, also welche Methode es besitzt. Sie beschreibt aber nicht, wie diese Fähigkeit umgesetzt wird. Sie ist damit eine klassische Schnittstelle.

Im Gegensatz zur Hierarchieschnittstelle kann eine Klasse beliebig viele Fähigkeitsschnittstellen (zusätzlich zu ihrer einen Oberklasse bzw. ihrer Hierarchieschnittstelle) besitzen. Eine andere Bezeichnung für Fähigkeitsschnittstelle ist Querschnittschnittstelle, weil sie querschnittliche Aufgaben abbildet.

Schnittstellen – Fähigkeitsschnittstellen



- Löst das Problem mit der darstellungsName() Methode
 - Eine Person ist **darstellbar**.
 - Eine Buchhaltung ist **darstellbar**.
- Beschreibt i.d.R. querschnittliche Aufgaben
- Anderer Name: Querschnittsschnittstelle
- Können auch Polymorph eingesetzt werden (Das „Etikett“ ist eine FS)



```

classDiagram
    class Dienst
    class Buchhaltung
    class Person
    class Darstellbar {
        <<interface>>
        darstellungsName() String
    }
    Dienst <|-- Buchhaltung
    Darstellbar <|-- Person
    Darstellbar <|-- Buchhaltung
            
```

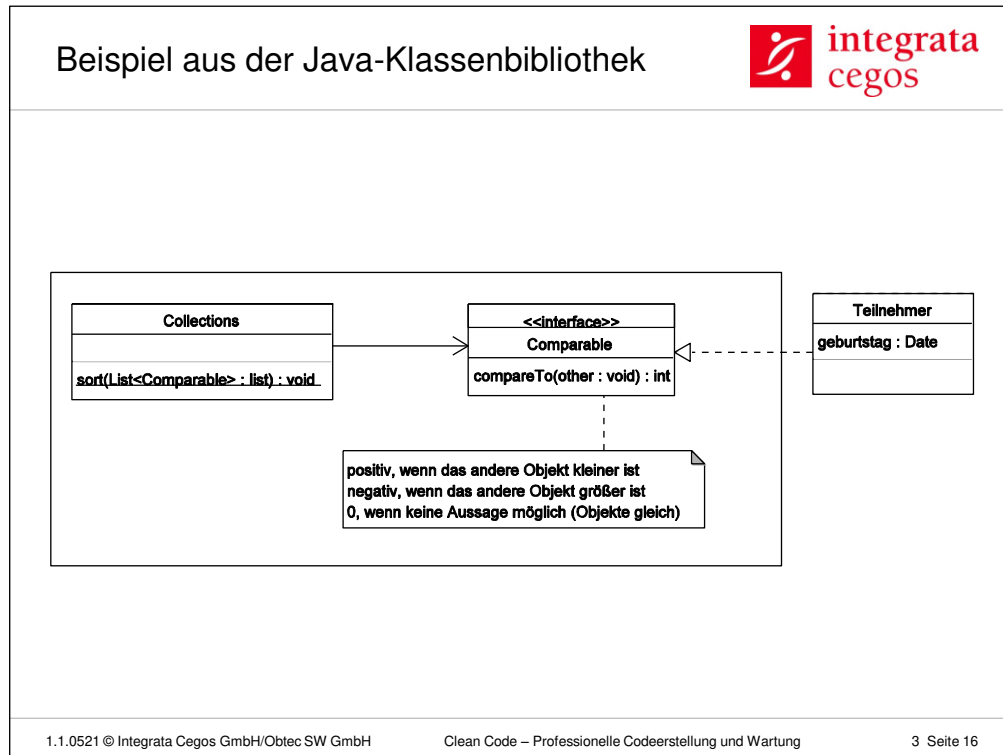
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code – Professionelle Codeerstellung und Wartung
3 Seite 15

Der große Vorteil an Fähigkeitsschnittstellen ist, dass auch sie polymorph eingesetzt werden können, d.h. eine (eher technische) Komponente, die das konkrete, fachliche Objekt überhaupt nicht kennen muss, kann dennoch mit allen Objekten, die dieses Interface implementieren, zusammenarbeiten. Das reduziert die Kopplung und erhöht die Wiederverwendbarkeit dieser Komponente ungemein.

In dem konkreten oberen Beispiel könnte man beispielsweise eine Klasse „ListenDruck“ erwarten, deren Aufgabe es ist, dem Nutzer eine Liste von Objekten auszudrucken. Statt nun für jedes ausdrückbare Objekt eine eigene Methode zu definieren, reicht es, eine einzige Methode `drucke(Liste<Darstellbar>)` zu erstellen.

Ein anderes, konkretes Beispiel aus der Java-Klassenbibliothek (leicht modifiziert):

*Das Interface **Comparable** beschreibt die Fähigkeit „vergleichbar“, d.h. jedes Objekt einer Klasse, die dieses Interface implementiert, ist mit anderen Objekten derselben Klasse vergleichbar (bezüglich einer Reihenfolge, der natürlichen Ordnung). Die einzige Methode des Interfaces, `compareTo(Object)` liefert nur mit einer Zahl zurück, welches von beiden Objekten kleiner ist. Kann man alle Elemente einer Liste paarweise mit einander vergleichen, so kann man damit effektiv die Liste auch sortieren, was durch die statische Methode `Collections.sort(List)` auch ermöglicht wird. Durch das Auslagern dieser querschnittlichen, nicht fachlichen Fähigkeit in ein eigenes Interface kann man jetzt mit minimalem Aufwand Listen von eigenen, fachlichen Objekten sortieren.*




Regel 3-7: Querschnittliche Fähigkeiten werden über Fähigkeitsschnittstellen realisiert. (Wied, Vers, Test)

Fähigkeitsschnittstellen sind also ein äußerst nützliches Hilfsmittel und umgehen dabei die Probleme von Mehrfachvererbung, da jede Methode nur einmal implementiert sein kann.

3.3.3 Mix-Ins


Der Nachteil von Fähigkeitsschnittstellen ist, dass dadurch der duplierte Code noch nicht verschwindet. Besteht die Schnittstelle nur aus disjunkten Methoden (bzw. aus einer einzigen Methode), so ist die Implementierung dieser Methode sicher vom eigenen Objekt abhängig.

Schnittstellen – Mix-Ins	 integrata cegos
<ul style="list-style-type: none"> ▪ Kombinieren abstrakte Fähigkeitsmethoden mit konkreten Methoden, die darauf aufbauen ▪ Methoden wie <i>equals()</i>, <i>isGreaterThan()</i>, <i>isLessThan()</i>, <i>isGreaterOrEqual()</i> etc. lassen sich alle auf die Methode <i>compare()</i> zurückführen. ▪ Gefahr der Mehrfachvererbung bleibt ▪ Durch das Abzielen auf nicht-fachliche Aufgaben ist die Gefahr einer Überschneidung deutlich geringer 	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
3 Seite 18	

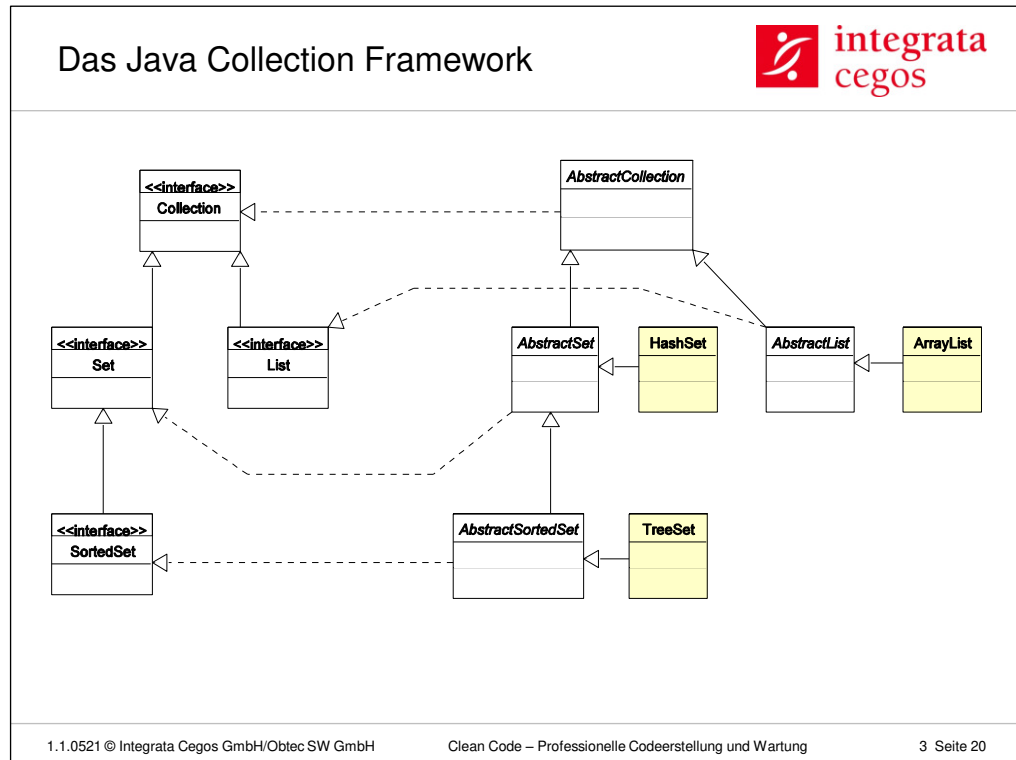
Nicht verschwiegen werden darf allerdings, dass natürlich bei Mix-Ins die Gefahren der Mehrfachvererbung zumindest wieder denkbar sind, auch wenn sie deutlich geringer sind, da wir es hier ja nicht mit fachlichen, sondern mit querschnittlichen Schnittstellen zu tun haben.

3.3.4 Parallele Schnittstellen-Hierarchien

Eine besondere Alternative zu den Hierarchieschnittstellen stellen die parallelen Hierarchien dar. Hierbei wird jede Vererbung sowohl mit Schnittstellen als auch mit Implementierungen abgebildet.

Parallele Schnittstellen Hierarchien	
<ul style="list-style-type: none">▪ Struktur für komplexe Hierarchien (i.d.R.) Frameworks▪ Vererbungsbäume werden abgebildet als Schnittstellen-Bäume▪ Gleichzeitig werden die Bäume mit konkreten (i.d.R. abstrakten Klassen) nachgebaut – diese Stellen Einsprung Punkte zur Vererbung dar▪ Konkrete Klasse erben von den abstrakten Klassen▪ nicht jedes Interface muss dabei eine konkrete (nicht abstrakte) Implementierung haben	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
3 Seite 19	

Der Vorteil ist eine saubere Architektur (modelliert wird nur mit dem Interface-Baum) und die Möglichkeit einer äußerst losen Kopplung, ohne die Fähigkeiten der Polymorphie aufzugeben.



Regel 3-8: Fachliche Vererbungen sollten als Parallele Hierarchien abgebildet werden. (Wied, Test)


Der Nachteil dieser Technik soll natürlich auch nicht verschwiegen werden: Die Anzahl der Klassen und Dateien steigt damit natürlich enorm an.

3.3.5 Client-Schnittstellen

Im Laufe der Entwicklung wachsen Schnittstellen naturgemäß immer weiter an. Neue Funktionalität, die ein Client benötigt, wird mit aufgenommen und die Schnittstelle immer schwergewichtiger. Irgendwann kristallisiert sich heraus, dass bestimmte Clients auch nur bestimmte Teile der Schnittstelle nutzen. Die Schnittstelle selbst verliert ihre Kohäsion.

3.3.6 Das Interface-Segregation-Principle ISP

Ein weiteres der Martin-Prinzipien ist das Schnittstellen-Abgrenzungs-Prinzip, das sich mit genau diesem Problem beschäftigt. Es lautet folgendermaßen:

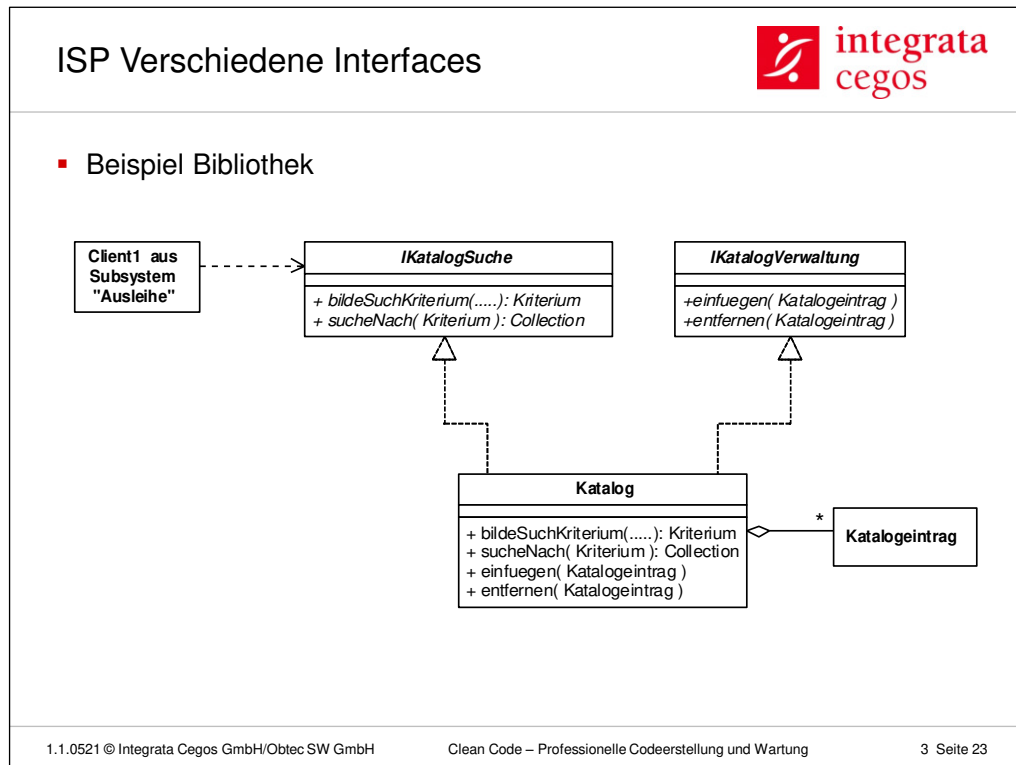
Interface Segregation Principle	
<ul style="list-style-type: none">▪ Clients sollten nicht gezwungen sein, sich auf Schnittstellen abzustützen, die sie nicht benutzen. (Regel 3-9) ▪ Lösung:<ul style="list-style-type: none">▪ Verschiedene Interfaces▪ Adapter Pattern	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
3 Seite 22	

Regel 3-9: Clients sollten nicht gezwungen sein, sich auf Schnittstellen abzustützen, die sie nicht benutzen. (Test, Vers, Wied)

Tun sie das nämlich doch, so koppelt man unweigerlich die unterschiedlichen Clients aneinander. Eine Änderung an einem Client kann durchaus eine Änderung des Server-Interfaces nach sich ziehen, eine Änderung, die sich damit auch wieder auf alle Clients auswirkt – was zumindest Kompilierzeit kostet, gerade wenn die verwendete Programmiersprache nur statisch verlinkt.

Verschiedene Interfaces

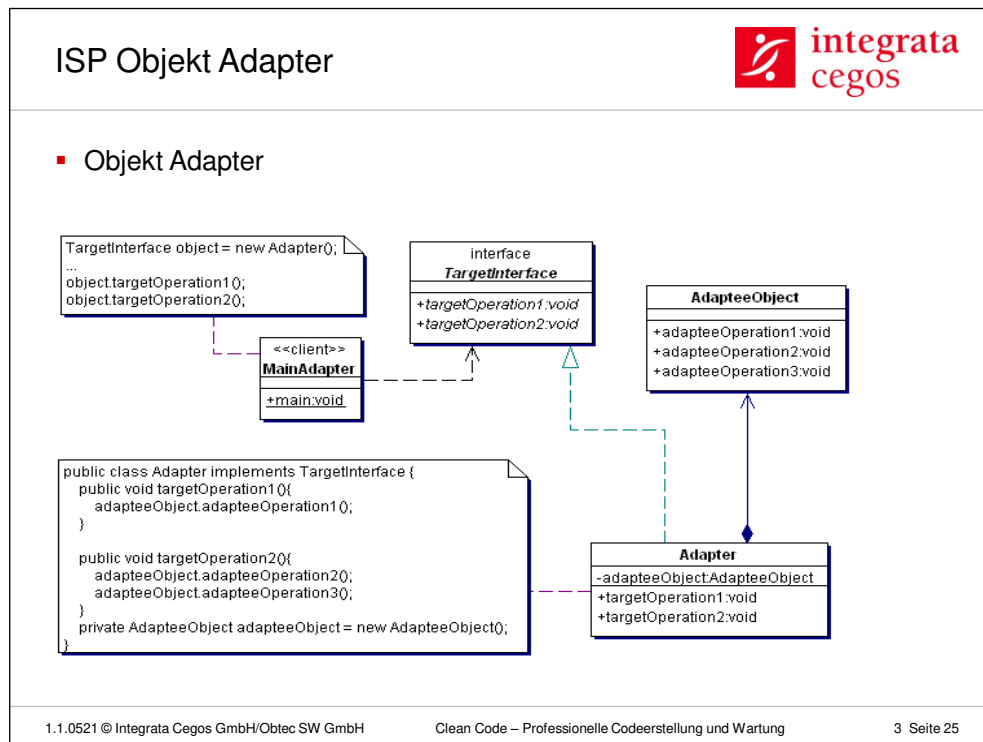
Die gesamte Schnittstelle einer Klasse kann **mehrere** Interfaces beinhalten, von der jede einen bestimmten *Aspekt* der Klasse ausdrückt. Ein bestimmter Client wird oft nur an *einem* Aspekt der Klasse interessiert sein – dieser wird durch ein Interface repräsentiert.



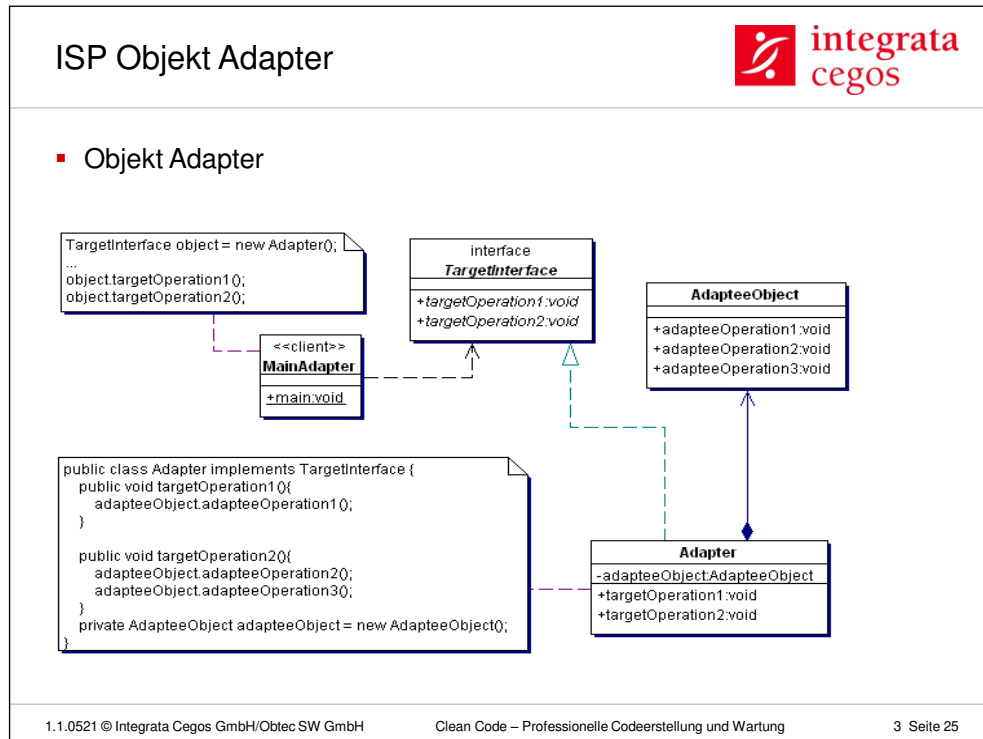
Ein Interface wird in diesem Fall vergleichbar mit einer **View** in der relationalen Welt eingesetzt.

Adapter Pattern

Das Adapter Pattern ist eine weitere Möglichkeit das Interface Segregation Principle umzusetzen. Das GOF Pattern gibt es zwei Varianten, den Objekt Adapter und den Klassen-Adapter:



Der Klassen-Adapter sieht folgendermaßen aus:



Aspekte

Verändert die Schnittstelle eines existierenden Objektes.

Wird auch Wrapper genannt.

Implementierung einer Schnittstelle unter Verwendung einer anderen (oder mehreren) Klasse.

Beliebige Klassen können unter Verwendung der Zielschnittstelle adaptiert werden (Wrapp-Mechanismus).

Klassenadapter verwenden (Mehrfach-)Vererbung – Objektadapter hingegen Komposition.

Varianten: Einweg-Adapter (ein adaptiertes Objekt) und Zweiweg-Adapter (mehrere adaptierte Objekte).

Rollen

TargetInterface – Schnittstelle an die andere Klassen angepasst werden.

Adaptee – Adaptierte Klasse / Objekt (nicht passende Schnittstelle)


Adapter – Die Klasse implementiert unter Verwendung von Adaptee das TargetInterface.

Client – fordert als Schnittstelle TargetInterface.

Regel 3-10: Das Interface-Segregation-Principle kann mit Hilfe des ADAPTER-Patterns umgesetzt werden. (Lesb, Vers, Wied)

3.4 Klassengrößen

In diesem Abschnitt beschäftigen wir uns mit der Größe einer Klasse. Hierfür definieren wir als erstes zwei Regeln:

Klassengrößen		
<p>Regel 3 -11: Klassen sollten klein sein. (<i>Lesb, Vers, Test</i>)</p> <p>Regel 3 -12 Klassen sollten noch kleiner sein. (<i>Lesb, Vers, Test</i>)</p>		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	3 Seite 27

Je kleiner eine Klasse ist, desto leichter ist sie zu überblicken und dementsprechend auch zu verstehen.

Dabei stellt sich die Frage, wie die Größe einer Klasse definiert werden kann. Ein trivialer Ansatz wäre, Zeichen zu zählen – was natürlich als Metrik vollkommen ungeeignet ist.

Die nächste Überlegung, eine Metrik, die lange Zeit recht beliebt war, ist die Anzahl der Zeilen (LOC – Lines of Code). Allerdings kann diese je nach Sourcecode-Formatierung schon deutlich abweichen. Eine etwas bessere Metrik sind Anweisungen, genauer Nicht-Kommentar-Anweisungen (NCSS – Non commenting source statements), diese sind genauer, aber aufwendiger zu zählen / zu berechnen.

Allerdings sind beide Varianten eher dazu geeignet, Methoden zu bewerten. Für Klassen empfiehlt sich eine andere Metrik: Die Anzahl der Verantwortlichkeiten. Die Nähe des Begriffs der Verantwortlichkeit zur Kohäsion sollte deutlich werden. Wenn alle Methoden fachlich zusammenhängen, dann wird die Klasse auch nur eine Verantwortlichkeit haben. Das ist aber nur zum Teil richtig, Verantwortlichkeiten können tatsächlich feiner sein, als fachliche Zusammenhänge.

Klassengrößen



- Mögliche Metriken
 - Zeilen (LOC)
 - Anweisungen (NCSS – Non commenting source statements)
- Besser: Verantwortlichkeiten
 - Wie viel macht die Klasse

3.4.1 Das Visions-Prinzip


Einen ersten Lackmus-Test für unsere Klasse erreichen wir mit Hilfe des Visions-Prinzips. Es lautet folgendermaßen:

Das Visionsprinzip		
<p style="text-align: center;"><i>Regel 3 -13</i></p> <p style="text-align: center;">Jedes Konzept (Pakete, Klassen, Methoden) muss sich verständlich in einem Hauptsatz (der Vision) beschreiben lassen. <i>(Vers, Wied)</i></p>		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung	3 Seite 29

Diese Beschreibung muss nicht alle Feinheiten beinhalten, sollte aber für sich alleine aussagekräftig genug sein. Ein typisches Zeichen für einen Verstoß gegen dieses Prinzip ist die Verwendung von Bindewörtern wie „und“, „oder“, „falls“ oder „außer“.

3.4.2 Das Single-Responsibility-Principle – SRP

Das Single-Responsibility-Principle ist ein weiteres Software-Engineering Prinzip:

Das Single-Responsibility-Principle (SRP)	
<ul style="list-style-type: none"> ▪ Fragestellung <ul style="list-style-type: none"> ▪ Was könnte mich dazu veranlassen, die Klasse zu verändern? ▪ Lösung <ul style="list-style-type: none"> ▪ z.B. <i>Eine neue Version wird ausgeliefert (d.h. die Versionsnummer ändert sich)</i> ▪ z. B. <i>Die Benutzeroberfläche ändert sich.</i> ▪ Lösung: Auslagern des Versionsteil in eine eigene Klasse Version ▪ Verantwortlichkeit wird also mit Änderungsgrund gleichgesetzt ▪ Konsequenz: <ul style="list-style-type: none"> ▪ Viele kleine Klassen 	
<div style="display: flex; justify-content: space-between; font-size: small;"> 1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH Clean Code – Professionelle Codeerstellung und Wartung 3 Seite 30 </div>	

Regel 3-11: (SRP) Für jede Klasse sollte es nur einen einzigen Grund geben, sie zu ändern. (Vers, Wied)

Verantwortlichkeit wird also mit Änderungsgrund gleichgesetzt

Anhand potentieller Änderungsgründe lassen sich die Methoden einer Klasse oft leichter aufteilen.

Natürlich führt das Zerlegen unserer Klassen in immer mehr kleine Klassen zu einer deutlichen höheren Anzahl an Gesamtklassen. Die Komplexität unseres Codes erhöht sich dadurch aber nicht. Vielmehr findet man sich eher leichter zurecht, da das Konzept einer Klasse besser durch den Namen beschrieben werden kann.

Wichtig ist dabei nur, dass man seine Klasse vernünftig organisiert, in Komponenten und Paketen.

Unser Ziel ist damit das folgende:

Ziel des SRP



Unser System besteht aus einer Vielzahl kleiner Klassen (statt weniger großer). Jede Klasse kapselt eine einzelne Verantwortlichkeit, hat nur einen einzigen, potentiellen Änderungsgrund und arbeitet mit wenigen anderen Klassen zusammen, um das gewünschte Verhalten abzubilden.

3.5 Änderungen ermöglichen

Es wird natürlich immer Gründe geben eine Klasse zu verändern, aber auch dafür gibt es Regeln im Software-Engineering.

3.5.1 Das Open-Closed-Principle – OCP)

Das Open-Closed-Principle beschäftigt sich mit der Problemstellung nachträglicher Änderungen an Klassen oder Modulen. Das Ziel sollte es sein, dass jede nachträgliche Änderung (Erweiterung) an einer Klasse nicht dazu führt, dass bestehender Code beeinflusst wird (bzw. dass dieser nicht einmal neu kompiliert werden muss).

Das Prinzip lautet:

Regel 3-12: (OCP) Klassen sollten offen für Erweiterungen, aber gesperrt für Veränderungen sein. (Wied, Wart)

Zerlegen wir diese Aussage in ihre zwei Bestandteile:

⇒ „Offen für Erweiterungen“

Das Verhalten der Klasse (des Moduls) muss erweiterbar sein. Das heißt, wir können die Klasse bei neuen Anforderungen so erweitern, dass die Änderungen eingearbeitet werden. Wir können also das Verhalten des Moduls ändern.

⇒ „Gesperrt für Veränderungen“

Das Erweitern des Verhaltens sollte nicht dazu führen, dass die Klasse selbst verändert wird. Im Klartext: weder der Sourcecode der Klasse, noch die aus dieser Klasse generierten Artefakte (Class- oder Object-Dateien, DLL- und Jar-Dateien, andere Bibliotheken) sollten durch die Erweiterung verändert werden.

Das Open-Closed-Principle



- Offen für Erweiterungen
 - Funktionalität muss darüber hinzugefügt/ergänzt werden können, dass Unterklassen geschrieben werden
- Gesperret für Veränderungen
 - Änderungen sollten nicht dazu führen, dass das Verhalten oder der Sourcecode der Klasse selbst geändert wird
- Konsequenz:
 - hinreichend Abstrakte Oberklassen schaffen

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code – Professionelle Codeerstellung und Wartung

3 Seite 33

Zusammen genommen stellen uns diese beiden Aussagen natürlich vor eine schwierige Aufgabe. Die offensichtlichste Art, das Verhalten einer Klasse zu ändern, nämlich die Klasse selbst zu modifizieren, wird durch die zweite Bedingung effektiv verhindert.

Der Schlüssel liegt natürlich in der Abstraktion. Wir erweitern unsere Klassen, indem wir von ihnen erben und verwenden umgekehrt in unserem Client-Code nur die abstrakten Oberklassen oder Interfaces (letzteres zieht sich ja durch alle Prinzipien).

Das bedeutet, wir müssen uns frühzeitig Gedanken darüber machen, ob eine Klasse überhaupt erweiterbar sein soll, und wenn ja, in welchen Methoden. Und diese Tatsache muss deutlich dokumentiert sein!

Umgekehrt muss verhindert werden, dass Klassen und Methoden, die nicht überschrieben werden sollen, das auch gar nicht zulassen dürfen⁸. Hierunter fallen zum Beispiel Methoden, die Invarianten (den Contract der Klasse) verletzen könnten.

Formulieren wir das als Regel:


Regel 3-13: Potentielle Oberklassen sollten verhindern, dass ihre Kind-Klassen jemals das Liskovsche Substitutionsprinzip verletzen können. (Wied, Wart)

⁸ Das wird zum Beispiel in Java mit dem Schlüsselwort final sichergestellt.

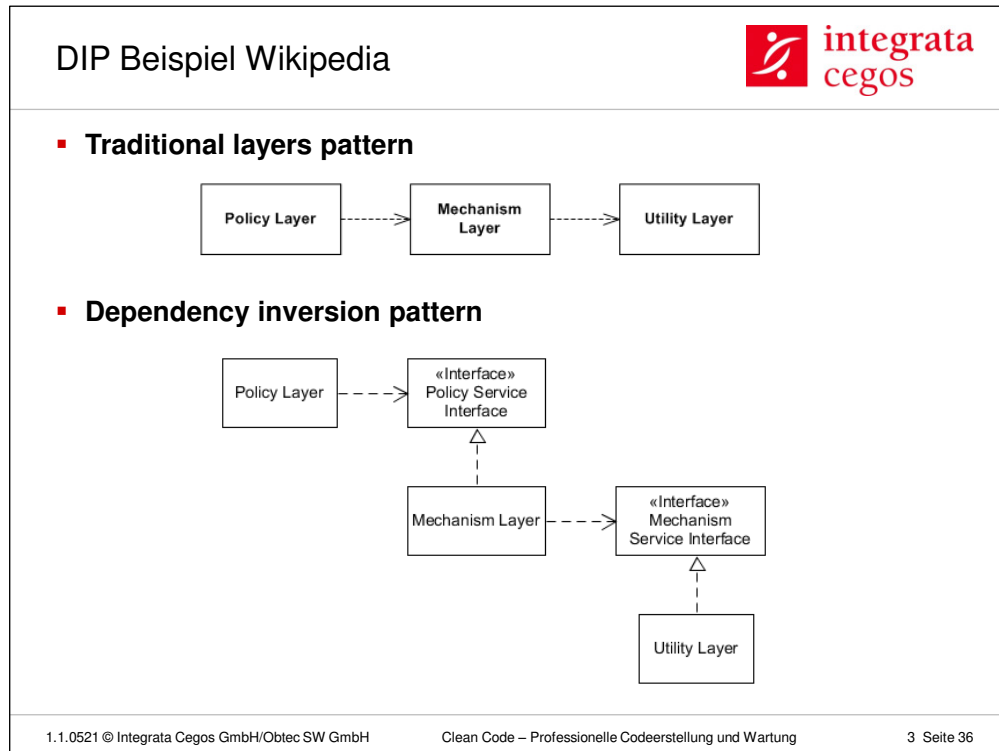
3.5.2 Das Dependency-Inversion-Principle – DIP)

Das Prinzip der umgekehrten Abhängigkeiten beschreibt, wie einzelne Klassen und Komponenten voneinander abhängen. In der „klassischen“ (prozeduralen) Softwareentwicklung gilt der Grundsatz, dass hochlevelige Module auf niedriglevelige Module zurückgreifen (ein Druckvorgang besteht aus dem Drucken von Zeichen und der Druckersteuerung).

Das Prinzip der umgekehrten Abhängigkeiten kehrt diese Ansicht um. Im Kern besteht es aus zwei Regeln:

Dependency Inversion Principle	
<p style="text-align: center;"><i>DIP Regel 3 -17</i></p> <p style="text-align: center;">Hochlevelige Module sollten nicht von niedrig-leveligen Modulen abhängen. Beide sollten nur von Abstraktionen abhängen</p> <p style="text-align: center;"><i>DIP Regel 3 -18</i></p> <p style="text-align: center;">Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen. (Wied, Wart, Test)</p>	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code – Professionelle Codeerstellung und Wartung
3 Seite 35	

Wo besteht nun das Problem? Die hochleveligen Module sind die Bestandteile unserer Software, in der die „interessanten“, fachlichen Entscheidungen getroffen werden, die niedrigleveligen diejenigen, die „in der Schlammzone“ die ganzen Details umsetzen. Ändert sich jetzt eines dieser Details, so hat das natürlich auch Auswirkungen auf die davon abhängigen Komponenten, als die hochleveligen Anteile unserer Software. Ein Implementierungsdetail ändert also möglicherweise unsere fachlichen Elemente! Umgekehrt wollen wir natürlich wenn möglich die hochleveligen Elemente auch wiederverwenden, denn diese enthalten ja den „interessanten“ Code – mit der klassischen Abhängigkeit können wir diese aber eben nicht von der Schlammzone trennen.



Diese Regel beschreibt natürlich im Großen und Ganzen, was wir weiter oben schon recht ausführlich diskutiert haben: Die Kopplung zwischen Modulen sollten nicht eng – über Klassen –, sondern lose – über Abstraktionen, also Interfaces – erfolgen. Der Leser sei hierzu insbesondere auf den Abschnitt zur Schnittstellenkopplung im vorherigen Kapitel verwiesen.

3.6 Zusammenfassung

Wir haben uns in diesem Kapitel mit den tiefergehenden Grundsätzen des Klassendesigns auseinandergesetzt. Dabei haben wir besprochen, wie Klassenhierarchien sinnvollerweise aufgebaut sein sollten, welche Grundsätze für die Größe einer Klasse an sich gelten und wie wir unsere Klasse offen für Veränderungen machen können.

In diesem Zusammenhang haben wir einige Prinzipien kennengelernt, die wir hier noch einmal aufzählen wollen. Die fünf Martin-Prinzipien lassen sich dabei mit dem Akronym SOLID auflisten, das letzte Prinzip (das Visions-Prinzip) ist davon unabhängig.

Wobei das wichtigste Prinzip K.I.S.S. heißt: Keep ist simple and stupid.

Zusammenfassung		
SRP	Single-Responsibility-Principle Für jede Klasse sollte es nur einen einzigen Grund geben, sie zu ändern.	
OCP	Open-Closed-Principle Klassen sollten offen für Erweiterungen, aber gesperrt für Veränderungen sein.	
LSP	Liskov-Substitution-Principle Unterklassen müssen an die Stelle ihrer Oberklassen treten können.	
ISP	Interface-Segregation-Principle Clients sollten nicht gezwungen sein, sich auf Schnittstellen abzustützen, die sie nicht benutzen.	
DIP	Dependency-Inversion-Principle Hochlevelige Module sollten nicht von niedrigleveligen Modulen abhängen. Beide sollten nur von Abstraktionen abhängen. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.	
Das Visions-Prinzip		
Jedes Konzept (Pakete, Klassen, Methoden) muss sich verständlich in einem Hauptsatz (der Vision) beschreiben lassen.		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH		Clean Code – Professionelle Codeerstellung und Wartung
		3 Seite 37

4

Namen

4.1	Einleitung.....	4-3
4.2	Welche Sprache?	4-3
4.3	Bedeutungsvolle Namen	4-5
4.3.1	Klassen	4-7
4.3.2	Abstrakte Klassen	4-7
4.3.3	Interfaces	4-8
4.3.4	Methoden	4-9
4.3.5	Konstruktoren	4-10
4.3.6	Namen und Kontexte	4-11
4.3.7	Besondere Namen	4-12
4.3.8	Missverständliche Namen	4-16
4.3.9	Textrauschen	4-17
4.3.10	Domänen-Sprache vs. Lösung-Sprache	4-19
4.3.11	Ein Konzept, ein Wort	4-19
4.3.12	Verwandte Konzepte.....	4-20
4.4	Namen und ihre Form.....	4-21
4.4.1	Groß- und Kleinschreibung	4-21
4.4.2	Optische Verwechslungen	4-22
4.4.3	Aussprechbare Namen	4-22
4.4.4	Typ- und Kontextbezeichner (encodings)	4-24
4.4.5	Wortspiele und „Slang“	4-26

4.5	Vorgehen	4-27
4.5.1	Ändern von Namen	4-27
4.5.2	Der Style-Guide	4-28
4.6	Zusammenfassung	4-29

4 Namen

4.1 Einleitung

Nachdem wir uns in den letzten Kapitel mit objektorientierten Grundsätzen beschäftigt haben, werden wir uns jetzt ein wenig davon lösen und uns mit Namen in unserem Programmcode beschäftigen.


Allen voran geht es hier natürlich um die Namen, die wir unseren Variablen geben, aber natürlich auch um die Namen für Methoden, Klassen und Komponenten.

Die Vergabe von guten Namen ist eines der wichtigsten Werkzeuge, das wir besitzen, um unseren Code lesbar und verständlich zu machen, und das mit in der Regel äußerst geringem Aufwand.

Wir werden uns im Folgenden mit einzelnen Prinzipien für gute und für schlechte Namen auseinandersetzen. Diese Prinzipien sind dabei weitestgehend eigenständig und unabhängig voneinander.

4.2 Welche Sprache?

Die erste Entscheidung, die zu treffen ist (bzw. die meist schon lange für uns getroffen wurde) ist die Frage, in welcher (menschlichen) Sprache soll unser Code geschrieben sein, und in welcher Sprache die Kommentare.

Welche Sprache	
<ul style="list-style-type: none">▪ Optionen für Teams in Deutschland<ul style="list-style-type: none">▪ Deutsch (bzw. Muttersprache des Teams)<ul style="list-style-type: none">▪ Begriffe näher an der Fachabteilung▪ Weniger Übersetzungsaufwand für Englisch-Unkundige▪ Schwierigkeiten bei möglichem Outsourcing▪ Englisch<ul style="list-style-type: none">▪ Schlüsselwörter der Programmiersprache sind in der Regel in Englisch (if, for, while, get... set...)▪ Gefahr: Statt schlechten lieber gar keine Kommentare▪ gemischt<ul style="list-style-type: none">▪ Nicht alles aus einem Guss▪ Nachträgliches Übersetzen ist unrealistisch	
<small>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH Clean Code - Professionelle Codeerstellung und Wartung 4 Seite 2</small>	

In der Regel bleiben eigentlich nur zwei Optionen: Die Muttersprache des Teams oder Englisch. Der Vorteil der Muttersprache besteht darin, dass die Entwickler ggf. weniger Energie in die Übersetzung ihrer Kommentare stecken müssen. Ist auf der anderen Seite Englisch gefordert, so könnte ein Entwickler zu der Einsicht kommen, lieber keine oder nur einfache Kommentare zu schreiben, statt sich die Blöße zu geben, falsches Englisch einzusetzen.

Umgekehrt gilt dieses Problem in der Zeit multinationaler Kooperation und Outsourcings natürlich auch für Team-Mitglieder deren Muttersprache eine andere ist.

Schwierig wird es insbesondere, wenn ein nationales Team erst später internationalisiert wird, weil zum Beispiel ein Teil der Softwarepflege in ein außer-europäisches Land outgesourced wird. Hier können im Zweifelsfall deutlich Mehrkosten für eine Übersetzung entstehen (die ja teilweise gar nicht mehr möglich ist); nachträglich Klassen und Methoden umzubenennen ist kaum möglich.

Ein weiterer Punkt der für Englisch spricht, ist die Tatsache, dass die meisten Programmiersprachen selbst (also die Schlüsselwörter wie z.B. `if`, `while` `goto`, ...) eben auch in „Englisch“ verfasst sind. Verwendet man englische Namen, so ist das Ergebnis näher an einem lesbaren Satz, als beim Wechsel zwischen den Sprachen. Das gleiche gilt für Sprachkonventionen wie Getter und Setter.


Im Normalfall ist das natürlich keine Entwicklerentscheidung, sondern wird in den Firmen- oder Projektrichtlinien festgeschrieben.

4.3 Bedeutungsvolle Namen

Der Name einer Variablen sollte aussagen, was diese Variable bedeutet bzw. wofür sie gedacht ist. Moderne Programmiersprachen lassen heute beliebige Längen für Variablennamen zu, und auch der zusätzliche Platzbedarf der Quelldateien ist kein Kriterium mehr.

Schauen wir uns ein Beispiel an:

Bedeutungsvolle Namen



```
for (int i = 1; i < m; i++) {  
    boolean b = true;  
    for (int j = 2; j < i; j++) {  
        if (i % j == 0) {  
            b = false;  
            break;  
        }  
    }  
    if (b) {  
        System.out.println(i);  
    }  
}
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

4 Seite 3

Was machen die einzelnen Variablen? Was macht der gesamte Code-Block? Ersetzen wir die Namen, ist der Algorithmus schon deutlich besser lesbar:

Bedeutungsvolle Namen



```
for (int possiblePrime = 1; possiblePrime < maxNumber;
possiblePrime++) {
    boolean isPrime = true;
    for (int possibleDivider = 2; possibleDivider <
possiblePrime; possibleDivider++) {
        if (possiblePrime % possibleDivider == 0) {
            isPrime = false;
            break;
        }
    }
    if (isPrime) {
        System.out.println(possiblePrime);
    }
}
```

Natürlich lässt sich dieser Code-Block noch weiter verbessern, aber dazu später mehr.

Regel 4-1: Variablen sollten Namen haben, die ihre Bedeutung widerspiegeln. (Lesb)

4.3.1 Klassen

Die objektorientierte Lehre verlangt, dass Klassennamen Substantive oder zusammengesetzte Substantive sind, also *Employee*, *Person* oder *Document*.


<h4>Namensregeln</h4> <ul style="list-style-type: none">▪ Klassen<ul style="list-style-type: none">▪ Substantive (Employee, Person, Document)▪ keine „Weichmacher“ (Manager, Service, Processor, Data, Info)▪ Abstrakte Klassen<ul style="list-style-type: none">▪ Echte Abstraktionen (die polymorph genutzt werden sollen)<ul style="list-style-type: none">▪ <i>Player ist eine abstrakte Oberklasse von Golfer und VideoGamer.</i>▪ <i>Car ist eine abstrakte Oberklasse von GasolineCar und ElectroCar.</i>▪ Gleiche Regeln wie für normale Klassen▪ Technische Hilfsklassen (nicht im Client sichtbar)<ul style="list-style-type: none">▪ Sollten mit dem Wort <code>Abstract</code> beginnen▪ Hierarchie-Interfaces werden wie Klassen benannt▪ Fähigkeitsschnittstellen werden mit Adjektiven benannt
<div>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH</div> <div>Clean Code - Professionelle Codeerstellung und Wartung</div> <div>4 Seite 6</div>

„Weichmacher“-Bezeichnungen wie *Manager*, *Service*, *Processor*, *Data* oder *Info* sollten möglichst nicht in einem Klassennamen vorkommen, diese deuten in der Regel auf einen Verstoß gegen das Single-Responsibility-Principle hin.

Implementiert eine Klasse direkt ein (Hierarchie-)Interface, so ist das Interface die zu benutzende Abstraktion, die Klasse trägt dann den Interface-Namen + „Impl“.

4.3.2 Abstrakte Klassen

Wir unterscheiden hier zwischen zwei Arten von Abstrakten Klassen: Echte Abstraktionen im fachlichen Sinn, also Oberklassen, die selbst nur nicht instanziiert werden können, aber im Client-Code polymorph verwendet werden (und teilweise auch als Ersatz für Hierarchie-Schnittstellen oder parallele Hierarchien verwendet werden, wenn die verwendete Sprache keine echten Interfaces kennt):

Player ist eine abstrakte Oberklasse von Golfer und VideoGamer.

Car ist eine abstrakte Oberklasse von GasolineCar und ElectroCar.

In diesem Fall gelten die gleichen Namensregeln wie für konkrete Klassen.

Die zweite Variante sind Implementierungsdetails, die für den Client-Code niemals sichtbar sind sondern dem Implementierer einer Schnittstelle als Ausgangspunkt dienen. Diese stehen in der Regel zwischen dem Hierarchie-Interface und den konkreten Implementierungen. Diese Klassen sollten im Namen deutlich machen, dass sie nur abstrakte Hilfskonstrukte sind. Das wird durch das Präfix „Abstract“ verdeutlicht.

Im Java Collection Framework gibt es zu dem Interface List eine Implementierung namens `AbstractList`. Diese erlaubt es, neue Listen zu implementieren, indem nur zwei Methoden realisiert werden (die in `AbstractList` abstrakt sind). Alle anderen (zahlreichen) Methoden von `List` sind bereits in `AbstractList` als Abstützung auf die beiden abstrakten Methoden realisiert.

Regel 4-2: Klassen und Abstraktionen tragen die Namen von (ggf. zusammengesetzten) Substantiven. (Vers)

Regel 4-3: Abstrakte Klassen als Implementierungshilfen sollten mit dem Präfix „Abstract“ versehen werden. (Lesb, Vers)

4.3.3 Interfaces

Zur Benennung von Interfaces haben wir in den vorangegangenen Kapiteln ja schon einiges gesagt. Fassen wir noch einmal zusammen:

(Parallele) Hierarchie-Interfaces sind Abstraktionen und folgen damit den Regeln für Klassen.

Regel 4-4: Fähigkeits-Interfaces und Mixins tragen Adjektive als Namen. (Vers)

Beispiele für Fähigkeitsinterfaces sind: `InterruptedException`, `Comparable`, `Serializable`, etc...

4.3.4 Methoden


Regel 4-5: Methoden sollten Verben oder aus Verben abgeleitete Bezeichnungen als Namen tragen. (Lesb, Vers)

Beispiele: `printPrimes()`, `postSave()`, `start()`

Besitzt die Methode ein einzelnes Argument¹, dessen Bedeutung nicht direkt aus dem Namen der Methode hervorgeht, so kann der Name um Beziehungswörter angereichert werden:

Beispiel:

Statt `printPrimes(int max)` kann man auch `printPrimesUpTo(int max)` schreiben. Der Aufruf liest sich dann eleganter: `printPrimesUpTo(15)`.

Methodennamen, Details	
<ul style="list-style-type: none">▪ Ist die Bedeutung eines Argumentes nicht klar, so kann die Bedeutung in den Namen hineinkodiert werden▪ Entscheidend ist die Verständlichkeit im Aufruf<ul style="list-style-type: none">▪ <code>printPrimes(int max)</code> ist verständlich▪ <code>printPrimes(15)</code> im Client aber weniger▪ Alternative <code>printPrimesUpTo(15)</code> ist klar verständlich	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 4 Seite 9

Regel 4-6: Zugriffsmethoden sollten mit `get`, `set` oder `is` anfangen. Andere Methoden sollten diese Präfixe nicht benutzen. (Lesb)

```
String name = participant.getName();  
employee.setSalary(500);  
if (printjob.isRunning()) ...
```

¹ Eine *monadische* Methode. Vergleiche hierzu auch das spätere Kapitel über Funktionen.

Grundsätzlich sollten Methoden aus Sicht des aufrufenden Clients benannt werden. Klingt die Methode in der Klasse selbst also ungewöhnlich (nicht falsch oder holprig) und dafür in einem AufrufszENARIO gut und „richtig“, so ist das vollkommen zu verschmerzen.

4.3.5 Konstruktoren

Natürlich ist der Begriff Konstruktor hier ein wenig ungewöhnlich, da wir ja in den meisten Sprachen den Konstruktor nicht umbenennen können. Stattdessen können wir aber statische Factory-Methoden (vgl. vorheriges Kapitel) verwenden, die dann wieder sprechende Namen haben können. Das ist umso wichtiger, wenn wir überladene Konstrukturen verwenden.

Konstrukturen


- Können Konstrukturen Namen haben?
 - Beispiel: Klasse Point mit kartesischem und polarem Konstruktor


```
Point upperLeft = new Point(10, 20);
Point lowerRight = new Point(10f, 20f);
```
 - Besser (FACTORY-Pattern):


```
Point upperLeft = Point.FromCartesian(10, 20);
Point lowerRight = Point.FromPolar(10f, 20f);
```
 - Noch bessere Alternative (aber mit zusätzlichem Aufwand: BUILDER-Pattern)


```
Point lowerRight =
    Point.buildWith().angle(10f).distance(20f).build();
```
 - oder


```
Point lowerRight =
    Point.buildWith().angleOf(10f).and().distanceOf(20f).andReturnIt();
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code - Professionelle Codeerstellung und Wartung
4 Seite 10

Beispiel:

Eine Klasse `Point` besitzt zwei Konstrukturen, einen, der den Punkt mittels kartesischer Koordinaten (also `x` und `y`) erstellt, und einen zweiten, der diesen Punkt mittels Polarkoordinaten (Winkel und Abstand zum Ursprung) erstellt.

Mittels Konstrukturen würde der aufrufende Code folgendermaßen aussehen:

```
Point upperLeft = new Point(10, 20);
Point lowerRight = new Point(10f, 20f);
```

Die Unterschiede zwischen den beiden Konstruktor-Aufrufen sind hier nicht wirklich eingängig. Nur das Vorhandensein des „f“ bei den Argu-

menten unterscheidet die beiden Aufrufe, aber das Ergebnis ist natürlich grundverschieden.

Mittels einer Factory-Methode sieht das ganze schon wesentlich besser aus:

```
Point upperLeft = Point.FromCartesian(10, 20);  
Point lowerRight = Point.FromPolar(10f, 20f);
```

Die Verwechslungsmöglichkeiten sind hiermit ausgeschlossen. Bleibt anzumerken, dass die zweite Zeile immer noch darunter leidet, dass die beiden Argumente vom gleichen Typ sind, aber ihre Reihenfolge nicht unbedingt eingängig ist (bei x und y ist das in der Regel besser). Eleganter könnte man hier durch die Verwendung des BUILDER-Patterns werden:

```
Point lowerRight =  
    Point.buildWith().angle(10f).distance(20f).build();
```

Das ist zwar ein wenig mehr Schreibarbeit, aber dafür völlig unmissverständlich.²

Regel 4-7: Unklare Konstruktoren sollten durch Factory-Methoden „benannt“ werden. Der Konstruktor selbst sollte dann nicht mehr sichtbar sein. (Lesb, Vers)

4.3.6 Namen und Kontexte

Namen stehen immer in einem bestimmten Kontext, und auch nur in diesem müssen sie gültig sein. Das Feld name kann vieles bedeuten, aber im Kontext einer Klasse Person ist seine Bedeutung klar. Kontext bedeutet in der Regel entweder Klasse oder Methode, seltener auch Schleifenkonstrukt.

Es besteht ein grundsätzlicher Zusammenhang zwischen der Größe des Kontexts und der Länge des Variablennamens: Je größer der Kontext, desto länger und eindeutiger muss auch der Name der Variablen sein. Oder umgekehrt: Für einen sehr kurzen und einfachen Kontext ist es durchaus erlaubt, auch einen kurzen Namen zu wählen (Beispiel: der Schleifenzähler).

Verlässt eine Variable den Kontext, so muss der Kontext dem Variablennamen beigefügt werden:


```
Person user = ...;  
String userName = user.getName();
```

² Alternativ könnte man die Builder Methoden noch „menschlicher“ formulieren und den Aufruf mit Bindewörtern (and() in diesem Fall) erweitern.

```
Point.buildWith().angleOf(10f).and().distanceOf(20f).andReturnIt()
```

4.3.7 Besondere Namen

Es gibt in der Softwareentwicklung einige Namen, die typischerweise immer gleich benutzt werden. Ein Beispiel dafür ist der Schleifenzähler *i*. Er ist so häufig und traditionell, dass wir in unbedenklich verwenden können. Warum haben wir das im Beispiel weiter oben nicht getan? Weil der Zähler darüber hinaus auch noch eine fachliche Bedeutung hatte.

Name (fortgesetzt)	
<ul style="list-style-type: none">▪ Namensregeln<ul style="list-style-type: none">▪ Namen sollten in ihrem Kontext verständlich sein▪ Verlässt eine Variable ihren Kontext, so sollte der Kontext (die „Herkunft“ in den Variablenamen einkodiert werden)<pre>Person user = ...; String userName = user.getName();</pre>▪ Je länger der Kontext, desto ausführlicher sollte der Name sein▪ Besondere Namen<ul style="list-style-type: none">▪ Schleifenzähler „i“<ul style="list-style-type: none">▪ Aber nur, solange der Schleifenzähler keine fachliche Bedeutung hat▪ Variable, die am Ende der Methode zurückgeliefert werden soll (Rückgabewert): „result“▪ durchlaufende Variable in for-Schleifen: „next“▪ Iteratoren: „it“	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 4 Seite 12

Als Faustregel gilt: Dient der Zähler nur zum Durchlaufen einer Datenstruktur, so ist in der Regel die Benutzung von *i* und *j* unproblematisch.

Sind Schleifen geschachtelt (was wir, wie wir später sehen werden, natürlich als erstes beheben wollen), so sind *i* und *j* eher unzweckmäßig, weil die Übersichtlichkeit hier ganz schnell leidet.

Andere Standardnamen werden sich einbürgern, sollten aber in den Programmierrichtlinien formuliert worden sein.

Einige Beispiele für derartige Konventionen:

- Der Rückgabewert, der im Laufe einer Methode berechnet oder zusammengebaut wird, heißt immer `result`.
- Die Variable, die beim Iterieren das jeweils nächste Element aufnimmt, lautet `next`.
- Der Iterator in einer Schleife heißt `it`.

```
public int sum(int[] values)
{
    int result = 0;

    for (int next : values) {
        result += next;
    }

    return result;
}
```

Regel 4-8: Eine Handvoll definierter Standardnamen erleichtert die Übersichtlichkeit, wenn sie *allen* Entwicklern bekannt sind. (Lesb)

Eine weitere Art von besonderen Namen sind zusammengesetzte. Nehmen wir eine Methode an, die aus einem Array von *Employee*-Objekten das Durchschnittsgehalt bestimmt. Wahrscheinlich wird die aufnehmende Variable in der Methode selbst den Namen `result` haben. Aber welcher Variablen im aufrufenden Code wird dieser Wert zugeordnet:

Ergebnisvariablen



- Wie könnte die Variable avg hier besser heißen?

```
int avg = averageSalary(employees);
```

- Möglichkeiten

- Prefix: averageSalaryOfEmployees
- Suffix: employeesAverageSalary
- Kurz: averageSalary
- Kurz, suffix: salaryAverage

- Erweiterter Prefix:

```
int averageSalaryOfPartTimeWorkers =  
averageSalary(partTimeWorkers);
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

4 Seite 14

```
int avg = averageSalary(employees);
```

Was wäre hier ein sinnvoller Name für `avg`? Es gibt einige denkbare Optionen:

- Prefix: averageSalaryOfEmployees
- Suffix: employeesAverageSalary
- Kurz: averageSalary
- Kurz, suffix: salaryAverage

Die Prefix-Lösung besticht durch die Tatsache, dass der der Variablenname fast genauso aussieht/klingt, wie der Methodenaufwurf, der das Ergebnis geliefert hat. Wichtig ist dabei, dass des Argumente-Teil nicht dem Typ der Argumente, sondern den tatsächlich übergeben Argumenten entspricht:

```
int averageSalaryOfPartTimeWorkers = averageSalary(partTimeWorkers);
```

Der Vorteil der beiden Suffix-Lösungen, dass das entscheidende fachliche Konzept (Employee bzw. Employee.salary) zuerst genannt wird, wird dadurch erkauft, dass es sprachlich ein wenig holprig klingt – ein Ziel, das wir später noch näher formulieren wollen, ist, dass sich unser Code fast wie Text liest. Die Suffix-Formen laufen diesem Ziel entgegen.

Die normale Kurzform (die eigentlich nur aus dem Methodennamen ohne seine Argumente besteht) ist natürlich kurz und kompakt, stützt sich

aber relativ stark auf ihren Kontext ab. Ist dieser eindeutig und hinreichend kurz, so ist die Kurzform natürlich vollkommen ausreichend.

Regel 4-9: Variablen, die das Ergebnis einer Methode aufnehmen, sollten den Namen dieser Methode tragen. Gibt es Verwechslungsgefahr, so sind dem Namen die Argumente des Aufrufs beizufügen. (Lesb)

Gelegentlich kann es notwendig sein, der Ergebnisvariablen auch noch den Namen der Instanz, auf der die Methode ausgeführt wurde, beizufügen.

Ergebnisvariablen mit Herkunft




- Es kann notwendig sein, auch die Herkunft in den Variablennamen einzukodieren:

```
public void createFamilyName(Person mother, Person father)
{
    String fatherLastName = father.getLastName();
    String motherLastName = mother.getLastName();
    return fatherLastName + "-" + motherLastName;
}
```

4.3.8 Missverständliche Namen

Ein Name sollte ein Konzept vermitteln. Passt der Name allerdings nicht zum Konzept oder deutet er sogar auf ein anderes Konzept hin, so wird die Name missverständlich.

Schlechte Namen



- Missverständliche Namen
 - Namen die auf ein falsches Konzept hinweise (z.B. eine Methode setXY, die Nebeneffekte hat)

```
public void setRunning() {
    controlThread.start();
}
```
 - Besser: startControlThread()
- Textauschen
 - Anhänge um „den Compiler zufrieden zu stellen“

```
public void addEvenValues(
    List<Integer> list1, List<Integer> list2) {
    for (Integer next : list1) {
        if (next % 2 == 0) {
            list2.add(next);
        }
    }
}
```
 - Besser: source und destination

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

4 Seite 17

Der Name `employeeList` für eine Variable sollte auch nur dann verwendet werden, wenn es sich tatsächlich um eine `List` (den Datentyp) handelt, nicht etwa bei einem Array oder einer eigenen Klasse. Besser wären Namen wie `employeeGroup` oder einfach nur `employees`.

Genauso sollte eine Methode auch nur dann mit `set` beginnen, wenn es sich dabei tatsächlich um einen Setter handelt. Das heißt nicht, dass unbedingt genau diese Variable unter der Haube gesetzt wird (das wäre ja ein Implementierungsdetail), aber die Semantik sollte einem Setter entsprechen:


```
public void setRunning() {
    controlThread.start();
}
```

Hier hat ein Programmierer zumindest noch die Chance zu erkennen, dass es sich bei der Methode nicht um einen Setter handelt, da die Signatur nicht passt (kein Argument). Bekäme `setRunning()` jetzt auch noch ein `boolean` Argument, wäre aber auch das dahin. Ein besserer Name wäre `startControlThread()`.

4.3.9 Textrauschen

Als Textrauschen bezeichnen wir Anhänge an Variablennamen, die wir nur deshalb verwenden, damit „der Compiler sich nicht beschwert“.

Schlechte Namen



- Weitere Beispiele für Textrauschen:
 - Pointer und Pointr
 - aPoint und thePoint
 - Person und PersonInfo
 - Payment und PaymentObject
- Domänen vs. Lösungssprache
 - Domänensprache ist die Sprache des Fachbereichs
 - Buchung, Rechnung, Posten, Rabatt
 - Lösungssprache ist die technische Sprache des Programmierers
 - Pattern, List, Controller, View

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung4 Seite 18

Ein häufiger anzutreffendes Rauschen sind Zahlen:

```
public void addEvenValues(  
    List<Integer> list1, List<Integer> list2) {  
    for (Integer next : list1) {  
        if (next % 2 == 0) {  
            list2.add(next);  
        }  
    }  
}
```

List1 und list2 haben hier unterschiedliche Bedeutungen. Bessere Bezeichnungen wären hier `source` und `destination` gewesen, damit wäre die Routine deutlich besser lesbar.

Eine Variante davon sind beabsichtigte Schreibfehler (zum Beispiel `Pointer` und `Pointr`), eine weitere Füllwörter wie „a“ und „the“ (was ist der Unterschied zwischen `aPoint` und `thePoint`?).

Eine andere Form von Textauschen sind inhaltlich bedeutungslose Wörter wie `Info` und `Data`. Was ist der Unterschied zwischen den Klassen `Person` und `PersonInfo`³? Wie unterscheiden sich `Payment` und `PaymentObject`.

Auch redundante Informationen sind unnötig. Den Namen einer Person als `NameString` zu bezeichnen ist eine solche Redundanz.

Um es noch einmal deutlich zu machen: Für alle oben genannten Fälle kann es Sinn machen, den Code doch so zu schreiben. Wenn aber einer der Punkte nur angewendet wird, um zwei Variablen voneinander zu unterscheiden, dann ist es Textauschen.

Regel 4-10: Die Unterschiede zwischen zwei gewählten Namen müssen so gewählt werden, dass der Leser sie inhaltlich versteht. (Lesb, Vers)

³ Natürlich mag es vollkommen begründete Situationen, in denen genau dieses Konstrukt verwendet werden sollte, aber diese Fälle sind dann API spezifisch und dort auch entsprechend dokumentiert.

4.3.10 Domänen-Sprache vs. Lösung-Sprache

Wenn wir Namen vergeben, müssen wir uns entscheiden, aus welcher Sprache wir unsere Namen wählen. Dafür gibt es zwei Möglichkeiten: Namen aus der Domänen-Sprache - also dem Fachbereich -, oder Namen aus der Lösungs-Sprache - grob gesagt unserer Programmiersprache mit Bibliotheken, Patterns und Konzepten.

Aus welcher Sprache sollten wir unsere Begriffe wählen? Klar ist, dass wir die wichtigen, fachlichen Klassen sicher in der Domänensprache schreiben werden. Auch die fachlichen Berechnungsmodule werden sicher fest in die Domäne eingebettet.

Die ganzen technischen Details allerdings sollten in der Lösungssprache formuliert werden. Benutzen wir einen *Observer* (ein weiteres Pattern), dann sollte die Klasse auch den Namen `xyObserver` bzw. `xyListener` tragen.

Grundsätzlich wird der Code schließlich von Programmierern und nicht von Angehörigen der Fachabteilung gelesen.

Regel 4-11: Fachliche Konzepte sollten in Domänen-Sprache, technische Details in der Lösungssprache formuliert werden. (Lesb, Vers)

4.3.11 Ein Konzept, ein Wort

Konzepte sollten durch ein einzelnes, durchgängig verwendetes Wort beschrieben werden. Wechselndes Verwenden von `get`, `retrieve` und `fetch`, um Objekte aus Datenstrukturen auszulesen, ist nicht nur verwirrend zu lesen, es erschwert auch das eigentliche Schreiben von Code.

Moderne IDEs bieten automatische Vervollständigung. Wissen wir also, dass wir aus unserer Datenstruktur ein Objekt mit `retrieve...` auslesen können, nicht aber wie die Methode genau heißt, liefern uns wenige Tastendrücke alle Methoden, die mit `retrieve` anfangen. Kennen wir allerdings nicht einmal den Anfang, so müssen wir zwangsläufig alle Methoden der Klasse durchsehen, bis wir die richtige gefunden haben.


Umgekehrt sollte ein Wort aber auch nur ein Konzept ausdrücken. Um bei dem obigen Beispiel zu bleiben: `retrieveFirst()` sollte nicht bei der einen Datenstruktur nur das erste Element zurückliefern, bei einer anderen aber das erste Element entfernen und zurückliefern.

Regel 4-12: Gleiche Konzepte sollten durch das gleiche Wort beschrieben werden, unterschiedliche Konzepte durch unterschiedliche Wörter. (Lesb, Vers)

4.3.12 Verwandte Konzepte

Oftmals finden wir in unserem Code verwandte Konzepte, die häufigste Verwandtschaft ist dabei der Gegensatz (zum Beispiel `add` und `remove`). Um diese Konzepte zu beschreiben, gibt es im Sprachgebrauch natürlich Wortpaare. Diese Paarbildung sollten wir auch im Code einhalten.

Heißt eine Methode, um einen Nutzer hinzuzufügen `addUser()`, so sollte die Methode, um ihn wieder zu entfernen `removeUser()` und nicht etwa `deleteUser()` heißen.

Konzepte																	
<ul style="list-style-type: none"> ▪ Gleiche Konzepte sollten immer mit dem gleichen Wort beschrieben werden <ul style="list-style-type: none"> ▪ <i>get</i>, <i>retrieve</i> und <i>fetch</i> sollten konsistentes, unterscheidbares Verhalten haben ▪ Unterschiedliche Konzepte sollten auch unterschiedliche Wörter nutzen <ul style="list-style-type: none"> ▪ <i>retrieve</i> sollte nicht in einer Methode ein Objekt zurückliefern und es in einer anderen löschen und zurückliefern ▪ Verwandte Konzepte besitzen in der Regel Wortpaare <table border="0"> <tr> <td><code>add/remove</code></td> <td><code>insert/delete</code></td> <td><code>begin/end</code></td> <td><code>lock/unlock</code></td> </tr> <tr> <td><code>show/hide</code></td> <td><code>create/destroy</code></td> <td><code>source/target</code></td> <td><code>start/stop</code></td> </tr> <tr> <td><code>min/max</code></td> <td><code>next/previous</code></td> <td><code>open/close</code></td> <td><code>old/new</code></td> </tr> <tr> <td><code>first/last</code></td> <td><code>up/down</code></td> <td><code>get/set</code></td> <td><code>get/put</code></td> </tr> </table> 	<code>add/remove</code>	<code>insert/delete</code>	<code>begin/end</code>	<code>lock/unlock</code>	<code>show/hide</code>	<code>create/destroy</code>	<code>source/target</code>	<code>start/stop</code>	<code>min/max</code>	<code>next/previous</code>	<code>open/close</code>	<code>old/new</code>	<code>first/last</code>	<code>up/down</code>	<code>get/set</code>	<code>get/put</code>	
<code>add/remove</code>	<code>insert/delete</code>	<code>begin/end</code>	<code>lock/unlock</code>														
<code>show/hide</code>	<code>create/destroy</code>	<code>source/target</code>	<code>start/stop</code>														
<code>min/max</code>	<code>next/previous</code>	<code>open/close</code>	<code>old/new</code>														
<code>first/last</code>	<code>up/down</code>	<code>get/set</code>	<code>get/put</code>														
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung																
4 Seite 20																	


Regel 4-13: Verwandte Konzepte sollten auch mit verwandten Begriffen beschrieben werden. (Lesb, Vers)

4.4 Namen und ihre Form

Nachdem wir oben auf die Bedeutung und den Inhalt eines Namens eingegangen sind, beschäftigen wir uns jetzt mit der Form, die der (mittlerweile hoffentlich bedeutungsvolle) Name annehmen sollte.

4.4.1 Groß- und Kleinschreibung

Folgen Sie bezogen auf Groß- und Kleinschreibung den Konventionen ihrer Sprache. Gibt es keine allgemeinen Konventionen, so sollten diese in den internen Programmierrichtlinien geschaffen werden. Selbst wenn diese bereits existieren, sollten sie sich trotzdem in den Richtlinien wiederfinden.

Namen	
<ul style="list-style-type: none">▪ Namen sollten den Konventionen der Programmiersprache folgen▪ Optische Verwechslungen<ul style="list-style-type: none">▪ durch verwechselbare Zeichen (kleines L, großes o)<pre>int a = 1; if (0 == 1) a = 0l; else l = 0l;</pre>▪ oder nur leicht unterschiedliche Namen<ul style="list-style-type: none">▪ XYZControllerForEfficientHandlingOfStrings▪ XYZControllerForEfficientStorageOfStrings	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 4 Seite 22

Als Beispiel hier die Konventionen für Java (C++ hat sehr ähnliche):

- Klassennamen beginnen mit einem Großbuchstaben, gefolgt von Kleinbuchstaben, Wortgrenzen werden durch einen weiteren Großbuchstaben kenntlich gemacht (sog. CamelCase).

`Person, Player, Document`

- Variablennamen beginnen mit einem Kleinbuchstaben und sind ebenfalls in CamelCase.

`document, numberOfPlayers`

- Konstanten werden ganz in Großbuchstaben geschrieben, wobei Wortgrenzen durch einen Unterstrich (_) deutlich gemacht werden.

MAX_VALUE, Sqrt_of_2

- Der Unterstrich und das Dollarzeichen sind zwar gültige Zeichen für einen Variablennamen, sollten aber nur in Ausnahmefällen benutzt werden.

4.4.2 Optische Verwechslungen

Regel 4-14: Namen sollten sich optisch so weit unterscheiden, dass man sie auf einen Blick auseinanderhalten kann. (Lesb)

Wir begründen diese Regel am besten mit einigen Gegenbeispielen:

Was tut folgender Code:

```
int a = 1;
if (0 == 1)
    a = 01;
else
    1 = 01;
```

Je nach verwendeter Schriftart muss man sich schon sehr dicht zum Code vorbeugen, um noch zu erkennen, was tatsächlich gemeint ist. Zu kurzen Variablennamen haben wir ja bereits einiges gesagt, hier sei nur noch erwähnt, dass das Problem sich noch massiv verstärkt, wenn die verwendeten Buchstaben das kleine L (Verwechslung mit der Ziffer 1) und das große O (Verwechslung mit der Ziffer 0) sind.

Das zweite Problem sind lange Namen, die sich nur in Nuancen unterscheiden: `XYZControllerForEfficientHandlingOfStrings` und irgendwo in einem anderen Modul die Klasse `XYZControllerForEfficientStorageOfStrings`⁴. Sicher ist dieses Beispiel etwas konstruiert, aber es soll ja auch nur das Prinzip zeigen.

4.4.3 Aussprechbare Namen

Wenn wir Code (oder Text⁵) lesen, neigen wir dazu, ihn im Kopf „vorzulesen“. Deshalb ist eines unserer ersten Ziele, dass sich unser Satz so einfach und ruhig wie möglich lesen lässt. Überraschungen und Dinge, die das Vorlesen ins Stocken bringen, führen dazu, dass wir vom Überfliegen des Codes in die konzentrierte, Wort-Für-Wort Leseart wechseln, die natürlich deutlich langsamer ist. Selbst wenn wir uns das Ausformulieren im Kopf abgewöhnen, tritt das Problem trotzdem wieder zu Tage, wenn wir mit anderen Entwicklern über den Code reden.

⁴ Beide Beispiele dieses Absatzes stammen aus „Clean Code“

⁵ Lassen wir Speed-Reading-Techniken mal außen vor.

Aussprechbare Namen



- Folgender Code ist relativ verständlich:

```
int mxNoPts = ...;
while (ptList.size() > mxNoPts) {
    Point rmvd = ptList.removeLast();
    sprList.add(rmvd);
}
```

- Besser ist allerdings

```
int maxNumberOfPoints = ...;
while (pointList.size() > maxNumberOfPoints) {
    Point removed = pointList.removeLast();
    spareList.add(removed);
}
```

- Extrem-Beispiel (aus produktivem Code!)

```
gaSuspSvcW0regLstnr()
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

4 Seite 23

Einer dieser Stolpersteine sind Abkürzungen. Betrachten wir folgenden Code:

```
int mxNoPts = ...;
while (ptList.size() > mxNoPts) {
    Point rmvd = ptList.removeLast();
    sprList.add(rmvd);
}
```

Die Namen haben alle eine Bedeutung, und nach einem Moment verstehen wir auch was der Code macht. Versuchen Sie einmal diesen Code laut vorzulesen. Bei jeder der Abkürzungen werden Sie wahrscheinlich ins Stolpern geraten und schließlich die abgekürzten Variablen entweder durch ihren vollen Namen („Max Number of Points“, das wäre die gute, aber unwahrscheinlichere Lösung), durch einen Ausspracheversuch („m'xnpts“ als Wort) oder durch Buchstabieren („m'x no Peh Teh Es“) ersetzen. In jedem Fall muss Ihr Gehirn zusätzliche Arbeit aufwenden.⁶

⁶ Und das war noch ein relativ harmloses Beispiel. Betrachten Sie folgendes (echtes) Beispiel: `gaSuspSvcW0regLstnr()` – falls Sie sich fragen: die Abkürzung steht für „Gather (oder get?) all suspended Services without registered listeners“


Versuchen Sie das gleiche mit der korrigierten Fassung:

```
int maxNumberOfPoints = ...;
while (pointList.size() > maxNumberOfPoints) {
    Point removed = pointList.removeLast();
    spareList.add(removed);
}
```

Regel 4-15: Namen sollten aussprechbar sein. Abkürzungen sollten nur in Ausnahmefällen verwendet werden, und auch dann nur sprechbare. (Lesb)


4.4.4 Typ- und Kontextbezeichner (encodings)

Früher war es allgemein üblich bzw. je nach Sprache sogar notwendig, den Typ einer Variablen in den Namen mit aufzunehmen, entweder als Präfix (die sogenannte Ungarische Notation: `iLength`, `sName`) oder explizit (`nameString`, `sizeInt`).

Schlechte Name	
<ul style="list-style-type: none"> ▪ Encodings: <ul style="list-style-type: none"> ▪ Ungarische Notation: <ul style="list-style-type: none"> ▪ <code>iLength</code> ▪ <code>sName</code> ▪ Explizit: <ul style="list-style-type: none"> ▪ <code>nameString</code> ▪ <code>sizeInt</code> ▪ Kontext encodings: <ul style="list-style-type: none"> ▪ <code>pNumber</code> (für Parameter) ▪ <code>fSize</code> (für Felder) ▪ Wortspiele und Slang <ul style="list-style-type: none"> ▪ <code>killThemAll()</code> ▪ <code>bigBang()</code> ▪ <code>call911()</code> ▪ <code>insertB4()</code> (statt <code>insertBefore()</code>) 	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung
4 Seite 24	

In modernen, stark-getypten Sprachen ist diese Zusatzinformation nicht mehr notwendig. Der Compiler fängt Zuweisungsfehler in der Regel frühzeitig ab, damit werden diese Zusätze zu Textauschen. Sie sind sogar gefährlich, wenn sich der Typ nachträglich ändert, die Änderung aber nicht auf den Namen ausgeweitet wird. So könnte eine Variable den Namen `phoneString` besitzen, tatsächlich aber vom Typ `PhoneNumber` sein ((Vergleiche hierzu auch 0

Ergebnisvariablen mit Herkunft



- Es kann notwendig sein, auch die Herkunft in den Variablennamen einzukodieren:

```
public void createFamilyName(Person mother, Person father)
{
    String fatherLastName = father.getLastName();
    String motherLastName = mother.getLastName();
    return fatherLastName + "-" + motherLastName;
}
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung4 Seite 16

).

Eine weitere, unnötige „Verzierung“ sind Kontextbezeichner, zum Beispiel „f...“ für Felder, „p...“ für Parameter und kein Präfix für normale lokale Variablen. Moderne IDEs besitzen die Fähigkeit, unsere Variablen je nach Kontext anders darzustellen (andere Farben oder Schriftstile), weshalb auch hier nur unnötig redundante Informationen transportiert werden. Das im vorigen Absatz angesprochene Problem mit nachträglichen Änderungen gilt hier im Übrigen analog. Wird durch Refactoring ein Parameter in ein Feld umgewandelt (ein Vorgang, den die meisten IDEs heute auch automatisieren), so kann dabei schnell vergessen werden, den Name anzupassen.

Das Ergebnis ist leider oft: nach einer Weile stimmen die Encodings sowieso nicht mehr und werden ignoriert. Ganz von der Tatsache abgesehen, dass sie eben auch gegen die Regel der Aussprechbarkeit verstoßen.

Regel 4-16: Encodings für Typen und Kontexte sollten nicht benutzt werden. (Lesb)

4.4.5 Wortspiele und „Slang“

In gewachsenem Code findet man des Öfteren Methoden mit Name wie: `killThemAll()`, `bigBang()` und `call911()`⁷. Diese Namen scheinen auf den ersten Blick ja ganz witzig, aber was sagen sie über unserer Professionalität aus?

Außerdem ist nicht gesagt, dass ein (gut englisch-sprechender) Mitarbeiter notwendigerweise direkt erkennen kann, dass `call911()` eine Fehlermeldung an den Administrator versenden soll.

Auch Mode-Sprachen wie Leetspeak sollte man dringend vermeiden (`insertB4()` statt `insertBefore()`).

⁷ Alles tatsächliche Beispiele

4.5 Vorgehen

Ein häufiges Argument, das gegen die obigen Regeln angeführt wird, ist, dass sie zu viel mehr Code (also Speicherplatz) und Tipparbeit führen würden.

Der zusätzliche Speicherbedarf ist nicht zu leugnen, allerdings fällt er bei den heutigen Speichermedien nicht mehr wirklich ins Gewicht.

Und die zusätzliche Tipparbeit wird uns durch moderne IDEs fast vollständig abgenommen. Man muss einen komplexen Namen in der Praxis nur genau einmal tippen, und das zu dem Zeitpunkt, an dem man sich ja sowieso überlegen sollte, wofür die Variable oder Methode eigentlich gedacht ist.

Tatsächlich kann es durchaus sinnvoll sein, zwei bis drei Stunden in eine interne Schulung zur effizienten Nutzung der IDE zu investieren.

4.5.1 Ändern von Namen

Was ist zu tun, wenn ein Name nicht mehr passt? Formuliert man die Frage so, ist die Antwort einfach: man ändert ihn. Leider zeigt die Praxis, dass man hier nur sehr zögerlich voran geht.

Grundsätzlich hat das Ändern eines Namens zwei Konsequenzen: Zum einen bedeutet das Ändern selbst Aufwand, um ihn an allen Stellen richtig zu ändern, zum Anderen müssen sich die Nutzer dann wieder an einen neuen Namen gewöhnen.

Beide Konsequenzen sind in der Praxis aber vernachlässigbar. Das Umbenennen mit allen Nebeneffekten (ein Refactoring) erledigt die IDE für uns, und die wenigsten Programmierer merken sich die tatsächlichen Namen. Stattdessen stützen sie sich vielfach auf Code-Ergänzungen ihrer Entwicklungsumgebung ab, in dem sie sich einfach alle anwendbaren Methoden als Auswahl anbieten lassen.

Eine Ausnahme gibt es allerdings (die wir in einem vorherigen Kapitel schon erwähnt haben). Methoden einer öffentlichen Schnittstelle dürfen natürlich nicht einfach umbenannt werden⁸. Deshalb ist bei der Wahl der Namen einer Schnittstelle natürlich größere Sorgfalt angebracht.

Sollte es dennoch notwendig werden, eine Methode umzubenennen, gibt es natürlich Werkzeuge. Damit werden wir uns später noch auseinandersetzen.

⁸ Es sei denn, die Schnittstelle hat unsere Entwicklergruppe noch gar nicht verlassen, dann ist sie nicht wirklich „öffentlich“.

Argumentationshilfen



- Namen sind leicht und einfach zu ändern
- Speicherbedarf für längere Namen fällt i.d.R. nicht ins Gewicht
- Namen die nicht mehr passen, sollten sofort angepasst werden
 - Aber natürlich Vorsicht bei Schnittstellen
- Durch Code Completion in modernen IDEs muss ein langer Name nur genau einmal geschrieben werden, danach ergänzt die DIE

Entscheidend für lesbaren Code ist ein Projekt-Styleguide!!!

4.5.2 Der Style-Guide

Entscheidend für sauberen, lesbaren Code ist eine Firmen- oder Projekt-interne Richtlinienammlung, die die Regeln, die für die Softwareerstellung gelten, zusammenfassen. Ein guter Anfang dazu sind die Regeln dieser Unterlage.

4.6 Zusammenfassung

In diesem Kapitel haben wir uns mit einem der wichtigsten Grundsätze von lesbarem (also professionellem) Code beschäftigt: den Namen. Es gibt wenig Dinge, die einen Code so schnell unleserlich machen, wie schlechte Namen.

Umgekehrt tragen gute Namen massiv zur Lesbarkeit eines Programmes bei.

5

Methoden

5.1	Einleitung.....	5-3
5.1.1	Was ist eine Methode?	5-3
5.2	Form	5-5
5.2.1	Länge.....	5-6
5.2.2	Blockgrößen.....	5-12
5.2.3	Namen	5-15
5.3	Inhalt.....	5-17
5.3.1	Eine Aufgabe	5-17
5.3.2	Die Vision.....	5-19
5.3.3	Abstraktionsebenen	5-19
5.3.4	Die Stepdown-Regel.....	5-19
5.4	Argumente	5-21
5.4.1	Niladische Methoden	5-22
5.4.2	Monadische Methoden.....	5-22
5.4.3	Dyadische Methoden	5-24
5.4.4	Triadische Methoden	5-25
5.4.5	Größere (Polyadische) Methoden	5-26
5.4.6	Flags	5-27
5.4.7	Ausgabe Parameter	5-29
5.4.8	Argument-Objekte.....	5-30
5.5	Stil	5-32

5.5.1	Seiteneffekte	5-32
5.5.2	Befehl oder Abfrage (Command Query Separation)	5-32
5.5.3	Mehrere Exit-Punkte	5-33
5.5.4	Rekursionen.....	5-34
5.6	Zusammenfassung	5-36

5 Methoden


5.1 Einleitung

In diesem Kapitel werden wir uns mit den Grundsätzen guter Methoden beschäftigen. Wir werden dabei auf Form und Inhalt genauso eingehen wie auf Strategien für Argumente und guten Stil.

Der Leser sei gewarnt, dass einiges in diesem Kapitel auf den ersten Blick extrem erscheinen wird – oder zumindest gewöhnungsbedürftig. Lässt man sich allerdings darauf ein, so wird man mit deutlich besser lesbarem Code belohnt.

5.1.1 Was ist eine Methode?

Was eine Methode in der Objektorientierten Programmierung darstellt, haben wir bereits im ersten Teil dieser Unterlage besprochen. Allerdings tauchen in der Literatur einige unterschiedliche Begriffe auf, die wir zumindest ein wenig auseinander ziehen wollen. Leider widersprechen sich die Definitionen teilweise, für unseren Fall verwenden wir diejenigen, die aus Sicht des Autors am meisten Sinn machen.

Begriffe	
<ul style="list-style-type: none">▪ Operation<ul style="list-style-type: none">▪ Die Schnittstelle/Signatur/Fähigkeit▪ Methode<ul style="list-style-type: none">▪ Die eigentliche Implementierungen▪ Funktion<ul style="list-style-type: none">▪ Eine Methode, die einen Rückgabewert liefert▪ Prozedur<ul style="list-style-type: none">▪ Eine Methode, die keinen Rückgabewert liefert (und damit einen Seiteneffekt hat)▪ Routine<ul style="list-style-type: none">▪ Begriff aus der Prä-OO Zeit	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	5 Seite 2

Operation stammt aus der OO Programmierung und beschreibt die Schnittstelle, also das, was derjenige, der die Operation aufruft, davon sieht.

Methode beschreibt die eigentliche Implementierung der entsprechenden Operation – also das, was im Quelltext tatsächlich steht.

Der Unterschied beider Begriffe wird deutlich, wenn man Polymorphie hinzuzieht: Einer Oberklasse `Parent` besitzt eine Methode `doIt()`, sowie drei Unterklassen, die alle die `doIt()` überschreiben. Dann gibt es **eine** Operation `doIt()` und vier Methoden, die diese implementieren.

Eine **Funktion** ist eine Methode, die einen Rückgabewert liefert. Je nach Definition kann man auch noch verlangen, dass die Funktion selbst Seiteneffektfrei ist.

Eine **Prozedur** ist dementsprechend eine Methode, die keinen Rückgabewert liefert, bzw. (in der schärferen Definition) einen Seiteneffekt hat.


Eine **Routine** schließlich ist ein Oberbegriff, der noch aus prä-OO Zeiten stammt.

Oftmals werden die Begriffe in der Praxis aber auch bunt durcheinander gewürfelt. Wir werden in dieser Unterlage in Zukunft in der Regel von Methoden sprechen und ggf. die Methodensignatur ansprechen. Gelegentlich werden wir auch den Begriff Funktion bemühen, und zwar in der schärferen Form (also keine Seiteneffekte).

5.2 Form

Zunächst wollen wir uns mit der äußeren Form von Methoden beschäftigen, also Größen, Namen etc.

Damit wir das aber vernünftig tun können, müssen wir zumindest zwischen zwei Arten von Funktionen unterscheiden:

Formen von Methoden	
<ul style="list-style-type: none">▪ Schnittstellen-Methoden<ul style="list-style-type: none">▪ Teil der öffentlichen/protected Schnittstelle▪ public oder protected▪ Abstraktions-Methoden<ul style="list-style-type: none">▪ Methoden, die dazu dienen, den Code besser lesbar zu machen, ohne die Schnittstellen zu verändern▪ Häufig als „Hilfsmethode“ bezeichnet	

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

5 Seite 3

Schnittstellen-Methoden sind Methoden, die in der öffentlichen Schnittstelle definiert sind. Sie besitzen die Sichtbarkeiten *public* oder *protected*, selten auch *package*.

Abstraktions-Methoden sind Methoden, die wir dazu nutzen unseren Code besser lesbar zu machen. Sie sind Implementierungsdetails der Schnittstellen-Methoden und haben daher die Sichtbarkeiten *package* und *private*. Eine häufig zu findende Bezeichnung für diese Art von Methode ist *Hilfsmethode*, wir werden diese Bezeichnung aber hier bewusst vermeiden, denn sie deutet auf eine Methode zweiter Klasse hin – eine Interpretation, von der wir uns im folgenden noch deutlich distanzieren werden.

Warum unterscheiden wir nun überhaupt zwischen diesen beiden Arten? Nun, der Grundgedanke der Kapselung ist, dass wir unsere Implementierungen jederzeit austauschen können.

Das heißt, wir können unsere Abstraktions-Methoden nach Herzenslust umbenennen, umformulieren oder auf andere Art anpassen. Das Gleiche gilt für die Implementierungen der Schnittstellen-Methoden. Beim Verändern der **Signatur** der Schnittstellen-Methoden sollten wir aber

äußerst vorsichtig vorgehen, schließlich hat das Auswirkungen außerhalb unseres Moduls.

5.2.1 Länge

Wir haben uns bei der Frage, wie groß eine Klasse sein soll, schon mit Metriken für die Größe beschäftigt. Bei einer Klasse haben wir die Verantwortlichkeiten gezählt, bei Methoden zählen wir stattdessen Zeilen (LOC) oder Anweisungen (NCSS) – bei gutem Methodendesign sollten beide Werte sowieso annähernd gleich sein.

Bei der Besprechung der Klassengrößen haben wir auch zwei Regeln aufgestellt. Die gleichen Regeln wollen wir auch für Methoden anwenden:

Regel 5-1: Methoden sollten klein sein. (Lesb, Test)

Regel 5-2: Methoden sollten noch kleiner sein. (Lesb, Test)


In der Literatur findet man sehr unterschiedliche Definitionen darüber, was ausreichend klein ist. Auch in der Praxis finden wir sehr unterschiedliche Konventionen.

Tasten wir uns an den Begriff ein wenig näher heran.

200 Zeilen sind sicher zu viel. Ein alter Grundsatz war, dass Methoden (bzw. Prozeduren) nicht länger als eine Bildschirmseite sein sollten. Das war in Zeiten, in denen Bildschirme eine feste Anzahl von Zeichen aufnehmen konnten (z.B. 80x24), in der heutigen Zeit, mit immer größeren Bildschirmen und immer höheren Auflösungen bekommt man aber durchaus auch 100 Zeilen auf einem Bildschirm unter.

Trotzdem sind die 20 Zeilen der alten Terminals erst einmal ein guter Wurf (ein paar Zeilen gehen ja für Editor-Funktionen, wie Zeilennummern, Menüs etc. verloren).

Was ist klein



- Wie groß soll eine Methode denn nun sein?
 - Obergrenze: eine Bildschirmseite
 - bei 1920x1200 Auflösung mit Schriftgröße 8px?
 - Guter erster Wurf: 20 Zeilen
- Das Hrair-Limit
 - Psychologisches Konzept
 - Ein Mensch kann nur maximal 7 ± 2 Konzepte gleichzeitig verarbeiten
 - Alles darüber wird unterbewusst gruppiert
 - Die genaue Zahl ist Veranlagung
 - Der Begriff „Hrair“ ist eine Homage von Grady Booch an den Roman „Watership Down“ von Richard Admas

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung5 Seite 6

5.2.1.1 Das Hrair-Limit

Gehen wir aber noch einen Schritt weiter. 20 Zeilen sind immer noch zu lang (wenn auch nicht immer vermeidbar!). Eine besserer Ansatz sind zwei bis vier¹ (oder auch im Extremfall sieben) Zeilen. Das klingt natürlich extrem, und um diese Struktur überhaupt erreichen zu können, müssen wir einige Grundsätze beherzigen, die wir im restlichen Kapitel besprechen wollen.

Woher stammt nun aber diese Zahl? Zum einen aus der Praxis – fast alles lässt sich auf diese Größe reduzieren (wie, werden wir noch sehen).

Eine andere Erklärung liefert uns die Psychologie – schließlich geht es ja um Menschen, die unseren Code lesen und verstehen sollen. Das Verständnis des Menschen für komplexe Modelle ist grundsätzlich begrenzt. 1956 hat der Psychologe George Miller einen Grundsatz formuliert, der besagt, dass ein Mensch nur in der Lage ist 7 ± 2 Entitäten oder Konzepte gleichzeitig zu verarbeiten. Zusätzliche Konzepte werden dann gruppiert und mit anderen Konzepten in Beziehung gesetzt. Die genaue Zahl ist von Mensch zu Mensch unterschiedlicher.

Versuchen Sie einmal, die Seiten eines normalen Würfels (also die Lage der Punkte zu beschreiben).

Haben Sie irgendwann damit begonnen, Gruppen zu bilden (die vier sind zwei mal zwei Punkte, die sechs besteht aus zwei Reihen mit je

¹ 2 – 4, das ist kein Schreibfehler

drei Punkten)? Das ist ein erstes Zeichen (aber natürlich kein Beweis!).

Der Informatiker Grady Booch (einer der drei Urväter von UML) nannte diese Grenze auch das *Hrair Limit*, ein Begriff, den wir in Zukunft auch verwenden werden.²

Auf unsere Methodenlänge bezogen bedeutet das, dass eine Methode mit mehr als 9 Zeilen (Rumpf, die Signatur lassen wir außen vor) von einem Menschen nicht mehr vollständig erfasst werden muss – er muss damit zusätzliche Energie (und Zeit) aufwenden, um die Methode zu verstehen.

Regel 5-3: Methoden sollten dem *Hrair-Limit* genügen (nicht mehr als 7 Zeilen) (Lesb, Test)

Bleibt natürlich der mögliche Einwand: Wenn wir statt weniger, großer Methoden viele kleine Methoden verwenden, haben wir das Problem damit nicht nur eine Ebene nach oben verschoben (also mehr Methoden geschaffen, als wir erfassen können)?

Die Antwort ist (hoffentlich nicht allzu überraschend) nein. Zum einen liegen unsere Methoden in verschiedenen Abstraktionsebenen vor (siehe **5.3.3 Abstraktionsebenen**, weiter unten), und wir betrachten immer nur eine Ebene gleichzeitig (die oberste Ebene ist dabei die Schnittstellen-Ebene), zum anderen werden wir beim konkreten Lesen in der Regel sowieso immer nur eine Methode einzeln betrachten.


Zu große Methoden haben wir behandelt. Aber was ist mit zu kleinen Methoden?

5.2.1.2 1-zeilige Methoden

Machen Methoden mit nur einer Zeile Sinn? Schließlich ist hier der formale Anteil (Signaturen und Klammer bzw. Bock-Schlüsselwort³) genau so groß oder sogar noch größer, als der eigentliche Code!

² *Hrair Limit* oder *Rule of Hrair* bezieht sich auf den britischen Literatur-Bestseller (der im englischsprachigen Raum häufig im Unterricht gelesen wird) „Watership Down“ (deutsch: „Unten am Fluss“). Die Hauptcharaktere sind dabei intelligente Hasen, die allerdings nur bis vier zählen können, alles darüber hinaus ist einfach „Hrair“ (für „viele“)

³ Z.B. BEGIN und END in Pascal

Extremfälle	 integrata cegos	
<ul style="list-style-type: none">▪ Einzeilige Methoden<ul style="list-style-type: none">▪ Zur Übersetzung zwischen technischen und fachlichen Methoden<ul style="list-style-type: none">▪ <code>registerUser(user)</code> statt <code>userList.add(user)</code>▪ Klarstellung einer unübersichtlichen Berechnung▪ Nullzeiler<ul style="list-style-type: none">▪ Machen nur bei Vererbung Sinn▪ Dabei sollte die Oberklasse die leere Methode beinhalten, die Unterklassen dann Code▪ Anders herum wäre ein Verstoß gegen das LSP▪ Nullzeiler sollten einen Kommentar enthalten, warum sie leer sind		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	5 Seite 8

Eine einzeilige Methode kann sehr wohl Sinn machen, wenn sie der Lesbarkeit dient. Häufig sind diese Einzeiler Berechnungen, die einen besseren Namen bekommen oder Brücken zwischen fachlichem und technischem Code (die *fachliche* Methode `registerUser()` führt in ihrer Implementierung zur *technischen* Umsetzung `userList.add(user)`).

5.2.1.3 Leere Methoden

Treiben wir das Ganze auf die Spitze. Machen leere Methoden Sinn? Wenn keine Vererbung im Spiel ist, sicher nicht. Gerade leere Methoden, die einfach „im Vorgriff“ eingefügt werden, gefährden die Verständlichkeit unseres Codes sehr deutlich (sie stellen außerdem einen Verstoß gegen das *Open-Closed-Principle* dar). Entweder, wir brauchen die Methode, oder wir brauchen sie nicht. Sollte aus irgendeinem (sicher nicht technischen Grund) eine Methode noch nicht implementiert werden können, die aber in der Schnittstelle bereits festgelegt ist, so sollte diese Methode niemals leer sein. Ähnlich schlimm ist es, sie nur mit einem TODO-Kommentar zu versehen. Warum? In beiden Fällen haben wir eine Diskrepanz zwischen dem, was die Methode behauptet zu tun (also ihrem Namen), und dem, was sie tatsächlich tut (nämlich nichts)⁴. Stattdessen legen wir ja eine Annahme fest: Diese Methode wird derzeit noch nicht aufgerufen, sie ist nur da, damit der Compiler zufrieden ist. So eine Behauptung sollten wir auch immer überprüfen, und eine harte Fehlermeldung werfen, wenn die Behauptung widerlegt wird. In unserem Fall sollte einfach die Methode eine Exception (oder ähnliches, je nach Programmiersprache werfen).

Bringen wir also eine Vererbung ins Spiel. Hier gibt es zwei Möglichkeiten:

Die Methode der Oberklasse ist leer, die Unterklasse hat Inhalt:

In diesem Fall erfüllt die Methode die Aufgabe einer abstrakten Methode, die nicht unbedingt überschrieben werden muss. Die Gefahr hierbei ist natürlich, dass die Vererbungshierarchie dadurch unscharf wird, da eine Oberklasse jetzt „unnötige“ Fähigkeiten besitzt.

⁴ Zugegeben, sollte die Methode `doNothing()` oder `noOp()` heißen, hätten wir eine andere Situation. Hier wäre schon die Schnittstelle unsinnig.

Sinnvolle Nullzeiler



- Lifecycle-Methoden
 - technische Methoden (eines technischen Objektes), die durch ein Framework / einen Container (im Gegensatz zum eigentlichen Client-Code) aufgerufen werden
 - können in einer abstrakten Oberklasse leer implementiert werden, um die konkreten Klassen kleiner zu halten
 - `init()`, `destroy()`
- Hook Methoden
 - Methoden, die von der Klasse selbst aufgerufen werden, um ein bestimmtes Verhalten zu garantieren (OCP)
 - Teil des Template-Patterns

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

5 Seite 9

Es gibt dafür zwei Standardanwendungsfälle, wobei beide naturgemäß eher technischer Natur sind:

Lifecycle-Methoden sind Methoden, die zu einem bestimmten Zeitpunkt im Leben eines Objektes aufgerufen werden, wobei die Objekte dabei in der Regel keine fachlichen Objekte sind. So könnte in einem offenen Framework beispielsweise eine (abstrakte) Klasse `SystemService` definiert sein, von der mehrere unterschiedliche Kindklassen erben (`DataBaseAccess`, `PrinterAccess`). `SystemService` besitzt drei Methoden: `init()`, `execute()` und `destroy()`. `init()` und `destroy()` sind dabei die Lifecycle-Methoden, die durch das Framework aufgerufen werden, wenn der jeweilige Dienst instanziiert bzw. beendet wird, also technische Methoden, `execute()` die Methode, die die eigentliche fachliche Aufgabe ausführt⁵.

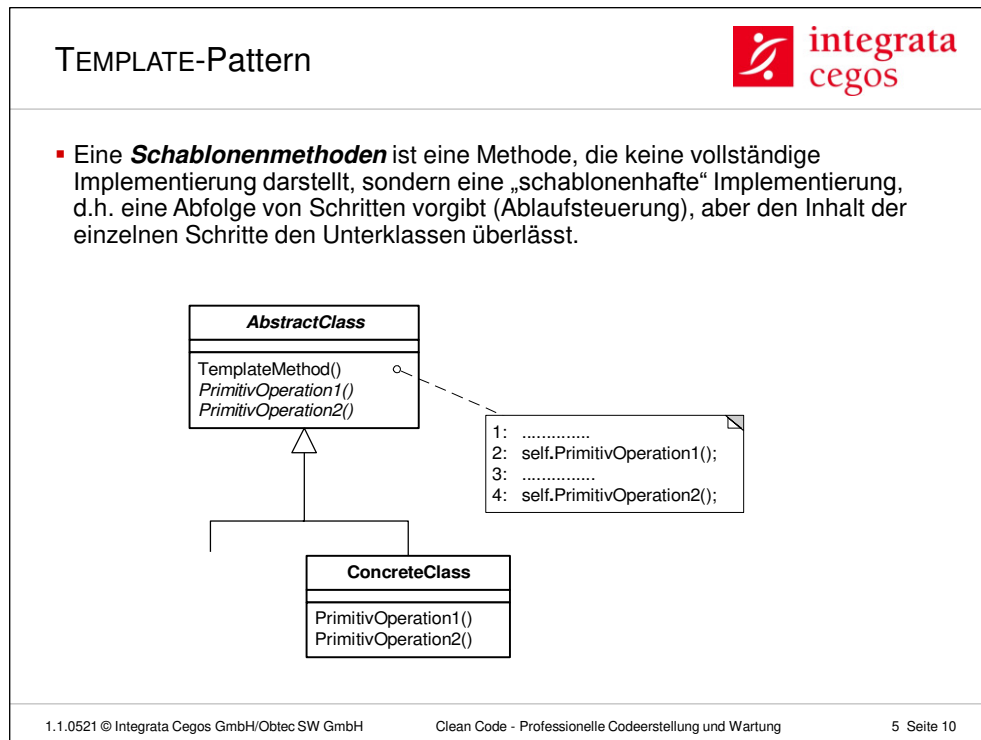
`DataBaseAccess()` könnte in `init()` eine Verbindung zur Datenbank herstellen. Benötigt `PrinterAccess` aber keine Initialisierung, so kann er einfach die leere Implementierung erben. Das reduziert den unnötigen Code in der Quelldatei.

Keinen Sinn macht es dagegen, die `execute()` Methode leer zu implementieren, da ein `SystemService` ohne fachlichen Auftrag wenig Sinn machen würde.

Hook-Methoden folgen einem ähnlichen Prinzip, nur werden hier die entsprechenden Methoden nicht von außen, sondern von der Klasse

⁵ Und damit eine Übersetzung zwischen technischen und fachlichen Komponenten darstellt.

selbst aufgerufen. Hier kommt das TEMPLATE-Pattern zur Anwendung, ein Pattern, das garantiert, dass Methoden in der richtigen Reihenfolge aufgerufen werden.



Die Methode der Oberklasse hat Inhalt, die Unterklasse ist leer:

Diese Konstruktion ist äußerst gefährlich. Die Oberklasse besitzt eine Fähigkeit, die die Unterklasse nach außen hin noch besitzt, die aber zu keinem Verhalten führt. Das bringt uns gefährlich nahe an einen Verstoß gegen das *Liskovsche Substitutionsprinzip*.

Grundsätzlich sollte man dieses Verhalten also vermeiden.

5.2.2 Blockgrößen

Die sinnvolle Länge von Methoden haben wir weiter oben schon besprochen. Wenn unsere Methoden aber nur höchstens sieben Zeilen haben dürfen, dann können auch unsere Blöcke nicht besonders groß sein.

Wir gehen einen Schritt weiter und formulieren „nicht besonders groß“ noch schärfer:


Regel 5-4: Blöcke sollten einzeilig sein. (Lesb, Test)

Das heißt, dass jeder Block immer nur aus einer einzelnen Anweisung bestehen sollte. Sind mehr als eine Anweisung nötig, so ist dafür eine eigene Methode zu schreiben.

Eine Konsequenz davon ist, dass unsere Methoden nie tiefer als einfach geschachtelt sind, was die Lesbarkeit und das Verständnis deutlich

verbessert. Außerdem schreiben wir jetzt in den Block hinein, was darin **fachlich** passiert, indem wir einen geeigneten Methoden-Namen wählen – was uns ggf. einen weiteren Kommentar einspart (mit Kommentaren werden wir uns im nächsten Kapitel noch ausführlich beschäftigen).

Einzeilige Blöcke



```

public void printPrimesUpTo(int maxNumber) {
    for (int possiblePrime = 1; possiblePrime < maxNumber; possiblePrime++) {
        boolean isPrime = true;
        for (int possibleDivider = 2; possibleDivider < possiblePrime; possibleDivider++)
        {
            if (possiblePrime % possibleDivider == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            System.out.println(
                possiblePrime);
        }
    }
}

public void printPrimesUpTo(int maxNumber) {
    for (int i = 1; i < maxNumber; i++)
        printIfPrime(i);
}

private void printIfPrime(int possiblePrime) {
    if (isPrime(possiblePrime))
        System.out.println(possiblePrime);
}

private boolean isPrime(int number) {
    for (int i = 2; i < number; i++)
        if (isDividerOf(i, number)) return false;
    return true;
}

private boolean isDividerOf (int i, int number) {
    return number % i == 0;
}

```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code - Professionelle Codeerstellung und Wartung
5 Seite 12

Schauen wir uns das Beispiel der Primzahlen aus dem letzten Kapitel noch einmal an. Zunächst noch einmal die Ursprungsversion (allerdings schon mit sprechenden Namen):

```

public void printPrimesUpTo(int maxNumber) {
    for (int possiblePrime = 1; possiblePrime < maxNumber;
possiblePrime++) {
        boolean isPrime = true;
        for (int possibleDivider = 2; possibleDivider < possiblePrime;
possibleDivider++) {
            if (possiblePrime % possibleDivider == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            System.out.println(possiblePrime);
        }
    }
}

```

Und jetzt die nach der obigen Regel umgestellte Version:

```
public void printPrimesUpTo(int maxNumber) {
    for (int i = 1; i < maxNumber; i++)
        printIfPrime(i);
}

private void printIfPrime(int possiblePrime) {
    if (isPrime(possiblePrime))
        System.out.println(possiblePrime);
}

private boolean isPrime(int number) {
    for (int i = 2; i < number; i++)
        if (isDividerOf(i, number)) return false;
    return true;
}

private boolean isDividerOf (int i, int number) {
    return number % i == 0;
}
```

Die zweite Variante ist deutlich besser lesbar. Und die längste Methode ist drei Zeilen lang!

Auch hier bleiben zwei Dinge anzumerken:

- In der Methode `isPrime()` haben wir natürlich getrickst, den eigentlich ist die Anweisung `if (isDividerOf (i, number)) return false` mehr als eine Anweisung. Hier haben wir aber direkt das erste Beispiel, in dem die Regel nicht bis zum äußersten anzuwenden ist – ein Exit-Punkt kann eben nicht in eine Methode ausgelagert werden.
- Die Forderung, jeden Block nur noch einzeilig aufzusetzen, erlaubt uns sogar, die geschweiften Klammern wegzulassen, ein zugegebenermaßen gewagter und gewöhnungsbedürftiger Schritt⁶.

Eine letzte Warnung ist allerdings angebracht:

Diese Art der Programmierung erzeugt sehr viele Methoden, damit sehr viele Methodenaufrufe. Das kann zu deutlichen Performance-Einbußen führen, wenn der Effekt nicht durch den Compiler (der hieraus einfach wieder große Methoden erzeugt), oder zur Laufzeit durch einen HotSpot-Compiler (wie zum Beispiel unter Java, der HotSpot-Compiler optimiert zur Laufzeit den Code, dabei bettet er unter anderem Methoden wieder in andere ein) verhindert wird.

⁶ Von der Benutzung von Blöcken ohne Klammern wird ja in fast jedem Styleguide abgeraten.

Andersherum: Mit einem optimierenden Compiler oder einem HotSpot-Compiler ist das Laufzeitverhalten beider Versionen (beinahe) identisch!

5.2.3 Namen

Wir haben uns ja bereits im letzten Kapitel ausführlich mit Namen beschäftigt. Wir wollen natürlich nicht alles wiederholen, sondern nur noch einmal den wichtigsten Grundsatz aufgreifen:

Namen von Methoden müssen beschreiben, was die Methode tut. Der Name kann dabei ruhig etwas länger sein, wir tippen ihn ja schließlich in der Regel nur einmal vollständig.

Ein guter Name für eine Funktion ist ein Verb oder ein Verb in Verbindung mit einem Substantiv. Das Substantiv dabei kann uns helfen, zu verstehen, was die Aufgabe des Arguments in der Methode ist.

Methodennamen	
<ul style="list-style-type: none">▪ Ein Verb oder ein Verb mit Substantiven<ul style="list-style-type: none">▪ <code>public void store(Order order)</code> ist verständlich.▪ Im Aufruf?<ul style="list-style-type: none">▪ <code>backend.store(priority);</code>▪ besser:<ul style="list-style-type: none">▪ <code>backend.storeOrder(priority);</code>▪ (Keyword-Form)	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 5 Seite 13

```
public void store(Order order)
```

Dieser Methodenname scheint gut verständlich zu sein, kann aber komplizierter werden, wenn der Name der übergebenen Variable nicht ganz eindeutig ist:

```
backend.store(priority);
```

Natürlich könnte man argumentieren, dass der Name für die Priority nicht besonders gut gewählt ist, aber was ist mit einer Routine, die eben nur zwei Bestellungen auf einmal bearbeitet, zum Beispiel eine reguläre

und eine priorisierte. Im Kontext dieser Methode sind die Namen „priority“ und „regular“ doch vollkommen ausreichend.

Eine Alternative wäre:

```
backend.storeOrder(priority);
```

Jetzt ist vollkommen klar, dass `priority` eine Bestellung sein muss. Man nennt diese Form des Methodennamens auch Keyword-Form. Noch wichtiger wird die Keyword-Form, wenn wir mehr als ein Argument haben und nicht logisch aus dem Namen hervorgeht, welches Argument was bedeutet.

`assertEquals(expected, actual)` ist ein klassisches Beispiel. Es handelt sich dabei um eine Methode aus dem JUnit-Framework, die für einen Testfall überprüft, ob das erwartete Ergebnis dem tatsächlichen entspricht. Die Reihenfolge der beiden Parameter wird aber in der Praxis ständig durcheinandergebracht. Hätte man die Methode `assertExpectedEqualsActual()` genannt, wäre die Reihenfolge vollkommen klar gewesen.

Regel 5-5: Der Name einer Methode muss zusammen mit seinen Argumenten auf *Client-Seite* verständlich sein. (Lesb, Test)

Auch bei Methodennamen können wir wieder die Holper-Regel anwenden. Finden wir nicht ohne allzu viel Mühe einen Namen, der unsere Methode beschreibt, so ist das ein Anzeichen dafür, dass unsere Methoden-Komposition noch nicht ganz zweckmäßig ist.

5.3 Inhalt

Nachdem wir die äußere Form beleuchtet haben, legen wir als nächstes unser Augenmerk auf den eigentlichen Inhalt unserer Methoden, also was diese nun eigentlich tun sollen. Dafür werden wir einige Regeln definieren, die uns unserem Ziel, gut lesbaren Code zu erstellen, noch näherbringen werden.


5.3.1 Eine Aufgabe

Ein Grundsatz der Objektorientierung ist die Trennung von Verantwortlichkeiten (*Separation of Concerns*) – ein Grundsatz der auch Eingang in das Prinzip der Kohäsion und das *Single-Responsibility-Principle* gefunden hat.

Wir wollen dieses Prinzip auch auf Methodenebenen anwenden und formulieren dafür die nächste Regel:

Regel 5-6: Eine Methode sollte eine Sache tun. Diese sollten sie gut tun. Diese sollten sie ausschließlich tun.⁷ (Lesb, Test)

Schauen wir uns noch einmal unser Beispiel an.

Eine Aufgabe pro Methode		
<ul style="list-style-type: none">▪ Wie viele Dinge macht diese Funktion: <pre>public void printPrimesUpTo(int maxNumber) { for (int i = 1; i < maxNumber; i++) printIfPrime(i); }</pre>▪ Möglichkeit 1: 2 Dinge<ol style="list-style-type: none">1. Durchlaufe alle Zahlen von 1 bis <i>maxNumber</i>2. Gebe jede Zahl aus, wenn sie eine Primzahl ist▪ Möglichkeit 2: 1 Ding<ol style="list-style-type: none">1. Gebe alle Primzahlen bis <i>maxNumber</i> aus.▪ Möglichkeit 2 beschreibt, was die Methode tut, Möglichkeit 1, wie sie es tut▪ Beide Varianten zusammen: <i>Um alle Primzahlen bis maxNumber auszugeben, durchlaufe alle Zahlen von 1 bis maxNumber und gib dabei jede Zahl aus, wenn sie eine Primzahl ist.</i>▪ Oder in English: <i>To printPrimesUpTo maxNumber, count from 1 to maxNumber and for each Number, printIfPrime.</i>		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	5 Seite 16

⁷ Frei übersetzt von Robert C. Martin, das Prinzip dahinter ist natürlich schon deutlich älter.

Wie viele Dinge macht die folgende Funktion?

```
public void printPrimesUpTo(int maxNumber) {  
    for (int i = 1; i < maxNumber; i++)  
        printIfPrime(i);  
}
```

Mögliche Antworten sind eins oder zwei. Wenn wir von zwei ausgehen, wäre das:

1. *Durchlaufe alle Zahlen von 1 bis maxNumber*
2. *Gebe jede Zahl aus, wenn sie eine Primzahl ist*

Gefühlt wäre natürlich die Lösung mit nur einer Aufgabe die bessere, denn sonst würde ja unsere (optisch doch eigentlich gute) Methode ja gegen die Regel verstoßen.

Wäre es nur eine Aufgabe, hieße die:

1. *Gebe alle Primzahlen bis maxNumber aus.*

Klingt das besser? Zumindest nicht schlechter. Die zweite Version beschreibt, **was** die Methode tut, die erste dagegen, **wie** sie es tut. Das sind zwei unterschiedliche Abstraktionsebenen.

Wir können beide Aufzählungen in einen „Um zu“-Satz bzw. Absatz unterbringen:

Um alle Primzahlen bis maxNumber auszugeben, durchlaufe alle Zahlen von 1 bis maxNumber und gib dabei jede Zahl aus, wenn sie eine Primzahl ist.

In Englisch haben wir dabei noch einen interessanten Effekt. Ein „Um zu“-Satz wird dann zu einem TO-Satz/Absatz⁸:

*To **printPrimesUpTo maxNumber**, count from 1 to **maxNumber** and for each Number, **printfPrime**.*

Der Name der Methode wird zu einem Teil des Satzes, genauso wie die einzelnen Anweisungen innerhalb der Methode.

Jetzt haben wir eine vollständige Beschreibung unserer Methode – ohne, dass wir noch eine weitere Form von Dokumentation brauchen.

⁸ Interessanterweise werden Methoden in der Sprache LOGO mit dem Schlüsselwort „TO“ definiert, was diese Art der Formulierung direkt in der Sprache motiviert.

5.3.2 Die Vision

Das Visionsprinzip haben wir bereits kennengelernt. Es besagt, dass sich ein Konzept in einem Satz ausdrücken lassen muss. Bisher haben wir es ausschließlich für Klassen genutzt, wir können es aber auch genauso für unsere Methoden heranziehen.

Allerdings bringt es hier wenig Neues ein, schließlich wollen wir ja sowieso schon unsere Methode mit einem „Satz“ beschreiben, nämlich dem Namen.

Weil die Vision ein echter Satz ist, kann sie uns aber helfen, unseren Methodennamen zu finden. Wir formulieren also erst die Vision und prüfen dabei schon, ob wir zu viel in der Methode tun wollen. Dann entwickeln wir aus der Vision den Methodennamen.

5.3.3 Abstraktionsebenen

Wir haben weiter oben festgestellt, dass die zwei Arten `printPrimesUpTo()` zu beschreiben unterschiedlichen Abstraktionsebenen entsprechen. Das ist das Prinzip jeder Methode. Sie stellt eine Anweisung auf einer höheren Abstraktionsebene dar und bricht sie herunter auf mehrere Anweisungen einer niedrigeren Ebene.

Dabei sind die oberen Ebenen in der Regel fachlich, die niedrigeren Ebenen eher technisch (auf hoher Ebene haben wir beispielsweise eine Methode `findBooksByAuthor()` die sicher auf niedrigerer Ebene irgendwann in Aufrufe einer Listenklassen herunter gebrochen wird -oder in Datenbankaufrufe etc...).

Ein Zeichen dafür, in welcher Abstraktionsebene wir uns befinden, sind damit auch die Objekte, die wir benutzen, also auf höherer Ebene nur fachliche Datenobjekte (Buch, Autor), auf niedrigerer Ebene die technischen Entsprechungen (Objekt, Liste).

Definieren wir eine weitere Regel:

Regel 5-7: Jede Methode sollte nur auf einer Abstraktionsebene agieren. (Lesb)

Ein Prüfstein dafür ist die Anwendung des „TO“-Satzes. Klingt dieser nicht schlüssig, so sind wahrscheinlich mehr als eine Abstraktionsebene beteiligt.

5.3.4 Die Stepdown-Regel

Haben wir alle obigen Grundsätze befolgt, können wir jetzt unseren Code fast wie einen Fließtext lesen, indem wir alle Methoden durch ihre TO-Sätze beschreiben, und dabei immer weiter in den Code hineingehen.

*To **printPrimesUpTo** **maxNumber**, count from 1 to **maxNumber** and for each Number, **printfPrime**.*


To **printlnPrime**, we check if the number *isPrime* and if so, print it on the console.

To check if a number **isPrime**, we check for all numbers between 2 and the number if **isDividerOf** number. If so, it is no Prime (false). Else, it isPrime.

Das liest sich mittlerweile recht verständlich. Probleme macht uns dabei noch die Methode `isDividerOf()`, die uns Schwierigkeiten bei formulieren macht. Diese Schwierigkeiten tauchen in der Regel dann auf, wenn Methoden mehr als einen Parameter haben und eigentlich als Operator fungieren (siehe dazu auch den nächsten Abschnitt). Lässt die Programmiersprache eine Funktion als Infix-Operator zu, so lässt sich diese natürlich eleganter formulieren.

Man könnte natürlich auch gut dafür argumentieren, die Methode `isDividerOf()` ganz wegzulassen, schließlich sollten der Ausdruck `number % i == 0` eigentlich verständlich genug sein.

Die Eleganz der Stepdown-Regel besteht dahin, dass der Leser beim durchlesen des Codes jederzeit entscheiden kann, aufzuhören oder noch tiefer hinein zu gehen, er aber immer einen sinnvollen Stand hat.

Methodenname	
<ul style="list-style-type: none"> Methodenname folgen dem Visionsprinzip, wobei der Methodenname selbst die (abgekürzte) Vision der Methode ist. Jede Methode agiert auf einer Abstraktionsebene <ul style="list-style-type: none"> Entweder sie arbeitet mit fachlichen Klassen <ul style="list-style-type: none"> <code>public Money calcShippingCostForOrder(Order order)</code> Oder mit technischen <ul style="list-style-type: none"> <code>private BigDecimal calcTotalCost(List<OrderItem> items)</code> <code>DataSource.openConnection()</code> Die oberste (Schnittstellen-Ebene) ist dabei i.d.R. rein fachlich Argumente sollten der Abstraktionsebene der Methode entsprechen <ul style="list-style-type: none"> <code>shipOrder(List<OrderItems> items)</code> mischt technische Parameter (List) und einen fachlichen Auftrag 	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung
	5 Seite 17

5.4 Argumente


Als nächstes wollen wir uns mit einem wichtigen Teil der Methoden beschäftigen, den Argumenten. Zunächst eine hoffentlich klare Regel:

Regel 5-8: Methodenargumente sollten auf der gleichen Abstraktionsebene liegen, wie die Funktion. (Lesb, Test)

Wir wollen Methoden im Folgenden anhand der Anzahl der Argumente unterscheiden. Ein Problem mit Argumenten ist, dass eine Menge Konzeptarbeit dahintersteckt. Je mehr Argumente, desto schwerer wird die Methode zu verstehen.

Die verständlichsten Methoden sind also Methoden ohne Argumente, dicht gefolgt von Methoden mit einem Argument. Zwei Argumente machen uns das Leben schon deutlich schwerer, drei Argumente sind kaum noch überschaubar und sollten vermieden werden. Und für Methoden mit mehr Argumenten bedarf es schon einer ausgezeichneten Begründung.

Auch was die Testbarkeit angeht, erhöhen mehr Argumente den Aufwand immens. Eine Funktion ohne Argumente und eine Funktion mit einem Argument erfordern nur wenige Tests, um alle Fälle abzudecken. Bei Methoden mit zwei Argumenten müssten wir für eine volle Testabdeckung alle möglichen *Kombinationen* der Werte abdecken. Bei drei und mehr Argumenten läuft das schnell ins Uferlose.

Anzahl der Argumente	
<ul style="list-style-type: none">▪ Niladische Methoden<ul style="list-style-type: none">▪ Sind beim Lesen am einfachsten zu erfassen<ul style="list-style-type: none">▪ <code>printResultInto(writer);</code>▪ <code>printResult();</code>▪ Erreicht wird das dadurch, dass der Parameter in ein Feld umgewandelt wird, dass vorher gesetzt wird▪ Monadische Methoden<ul style="list-style-type: none">▪ Das einzelne Argument übernimmt im durch den Namen der Methode gebildeten „Satz“ die Aufgabe des Objekts<ul style="list-style-type: none">▪ <code>printIfPrime(myNumber)</code>▪ <code>openFile("data.txt")</code>	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung
	5 Seite 20

5.4.1 Niladische Methoden

Niladische Methoden (Methoden ohne Argumente) sind aus Sicht der Lesbarkeit ideal. Das Konzept der Methode steckt vollständig in ihrem Namen, ohne dass man sich Gedanken über die Bedeutung der Argumente machen muss.

Betrachten wir ein Beispiel:

```
printResultInto(writer);
```

Was stimmt mit dieser Methode nicht? Zunächst mischt sie Abstraktionsebenen, den `Writer` ist wahrscheinlich eine technische Klasse, die Methode aber eher fachlich. Außerdem wird der `Writer` bei anderen Methoden derselben Klasse wahrscheinlich auch benötigt. Und meistens exakt dasselbe Objekt sein.

Besser wäre sicher:

```
printResult();
```

Wie kann man dieses Ergebnis aber erreichen? Natürlich, indem man die Argumente der Methode in Felder der besitzenden Klasse umwandelt bzw. eine Hüllklasse schreibt, die das Argument als Feld aufnimmt.

Natürlich lässt sich das nicht immer so einfach anwenden. Wir sollten trotzdem das Ziel vor Augen haben, wann immer möglich Niladische Methoden zu verwenden.

5.4.2 Monadische Methoden

Monadische Methoden (also Methoden mit einem Argument) sollten die häufigsten Methoden sein. Sie haben den Vorteil, dass sie in der Regel ohne Mühe lesbar sind, das einzelne Methoden-Argument erfüllt in unserem beschreibenden Satz die Aufgabe des Objektes (im grammatikalischen Sinne!).

Hauptarten von Monaden



- **Abfragen**
 - liefern Informationen zum Argument
 - `List.contains(Object o)`
 - `String.indexOf("Hallo")`
- **Transformatoren**
 - wandeln das Argument in ein anderes um
 - `InputStream openFile("data.txt")`
 - `List.get(15)`
- **Events**
 - ändern den Zustand des Objektes / Systems
 - Prominentestes Beispiel: `setter`
- Mischformen (ein Setter, der gleichzeitig etwas zurückgibt) sind schwerer lesbar.

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung5 Seite 21

`printIfPrime(myNumber)` druckt `myNumber`, wenn es eine Primzahl ist. `InputStream openFile("data.txt")` öffnet die Datei mit dem Namen `data.txt` und liefert eine `InputStream` auf diese Datei zurück.

Grundsätzlich gibt es zwei klassische Anwendungsfälle für Monaden:

- **Abfragen** liefern eine Information über das Argument zurück:
`List.contains(Object o)`, `String.indexOf("Hallo")`
- **Transformatoren** wandeln das Argument in ein anderes um. `InputStream openFile("data.txt")`, `List.get(15)`

Aus dem Namen der Methode sollte deutlich hervorgehen, um welchen Typ es sich handelt.


Eine dritte Form fehlt uns allerdings noch. Zu dieser gehören beispielsweise die Setter-Methoden. Es handelt sich um Methoden ohne Rückgabewert, die lediglich den Zustand des Objektes / des Systems ändern. Diese Form nennen wir **Event**.

Regel 5-9: Eine monadische Methode sollte immer eine Abfrage, ein Transformator oder ein Event sein. (Lesb, Vers)

Andere Formen erschweren die Lesbarkeit. Weiter oben haben wir bereits eine Möglichkeit gesehen, wie wir eine andere Form wie einen Transformator erscheinen lassen können (mittels Dopplung als Rückgabewert).

5.4.3 Dyadische Methoden

Dyaden, also Methoden mit zwei Argumenten, sind schwieriger zu handhaben als Monaden. Natürlich sind sie oftmals völlig verständlich, benötigen aber dennoch einen Moment des Innehaltens.

Dyadische Methoden	
<ul style="list-style-type: none"> ▪ Dyadische Methoden <ul style="list-style-type: none"> ▪ Sind schwerer zu erfassen als Monaden ▪ erfordern Sorgfalt bei der Reihenfolge der Argumente (Beispiel: <code>assertEquals()</code>) ▪ Führen leicht dazu, dass Argumente ignoriert werden ▪ Sind schwerer zu testen, da es mehr mögliche Kombinationen gibt, die getestet werden müssen <p style="text-align: center;"> <i>„Die Stellen, die wir ignorieren, sind die Stellen, an denen sich die Bugs verstecken“</i> <i>Robert C. Martin</i> </p>	
<div style="display: flex; justify-content: space-between; font-size: small;"> 1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH Clean Code - Professionelle Codeerstellung und Wartung 5 Seite 23 </div>	

`printResultField(fileWriter, "name")` ist komplizierter als `printResult("name")`. Natürlich nicht dramatisch, aber mit der Zeit wird der Leser lernen, den `fileWriter` einfach zu ignorieren. Und um R.C. Martin zu zitieren: „Die Stellen, die wir ignorieren, sind die Stellen, an denen sich die Bugs verstecken“.

Die Anzahl der Argumente bezieht sich hier allerdings (wie bei den anderen auch) nicht notwendigerweise auf die echte, gezählte Anzahl, sondern auf die logischen Argumente:


```
Point upperLeft = Point.FromCartesian(10, 20);
Point lowerRight = Point.FromPolar(10f, 20f);
```

In diesem Beispiel aus dem vorherigen Kapitel ist die erste Zeile eigentlich eine Monade! Die `x` und `y` Koordinate sind *geordnete* Element desselben Wertes (Wertepaars). Etwas schwieriger ist es in der zweiten Zeile, da hier (wie wir ja schon besprochen haben), eine strikte Ordnung nicht vorliegt. Es empfiehlt sich deshalb, die zweite Methode als Dyade zu betrachten.

Natürlich sind Dyaden in der Praxis nicht zu vermeiden – und sie stellen auch nicht die Wurzel allen Übels dar⁹. Allerdings sollten sie mit Bedacht eingesetzt werden. Und man sollte immer prüfen, ob es nicht eine bessere Alternative gibt.

5.4.4 Triadische Methoden

Triaden¹⁰, also Methoden mit drei Argumenten, sind äußerst gefährlich. Zum einen sind sie sehr schwer zu überschauen, zum anderen sehr schwer zu testen.

Triadische Methoden	
<ul style="list-style-type: none">▪ Sind schwer zu überblicken▪ Sind noch schwerer ausführlich zu testen▪ erfordern vom Leser Vorwissen oder Zeitaufwand▪ sollten wenn möglich durch Argument-Objekte oder Instanz-Variablen reduziert werden	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung
5 Seite 24	

Insbesondere die Reihenfolge der Argumente wird uns immer wieder dazu bringen, beim Lesen (und beim Schreiben) anzuhalten, und darüber nachzudenken. Und der Versuch, einen sinnvollen Namen ist von vorneherein fast immer zum Scheitern verurteilt.

Natürlich werden wir gelegentlich Triaden verwenden müssen, müssen dabei aber akzeptieren, dass diese fast immer Vorwissen des Lesers voraussetzen – oder eben querlesen erfordern.

⁹ Den die ist ja gemäß D.Knuth die „voreilige Optimierung“

¹⁰ Kein Bezug zur chinesischen Mafia

5.4.5 Größere (Polyadische) Methoden

Im Gegensatz zu Dyaden sind Polyaden tatsächlich die Wurzel allen Übels. Sie führen zu Code, der kaum noch erfasst werden kann (ohne ihn Anweisung für Anweisung auszuwerten), insbesondere wenn die Anzahl der Argumente das Hrair-Limit übersteigt.

Noch mehr Argumente (Polyaden)



- Ein extremes Beispiel:

```
GridBagConstraints(int gridx, int gridy, int gridwidth, int
gridheight, double weightx, double weighty, int anchor, int
fill, Insets insets, int ipadx, int ipady)
```

- Der Aufruf dazu ist noch schlimmer

```
new GridBagConstraints(5, 10, 1, 1, 1.0, 2.0, CENTER, BOTH,
emptyInsets, 1, 1)
```

- Einzige Chance: Kommentare

```
new GridBagConstraints(
    5, 10, // grid position
    1, 1, // cell size
    1.0, 2.0, // weight
    CENTER, // anchor
    BOTH, // fill
    emptyInsets,
    1, 1) padding
```

Eine der extremsten Beispiele aus den Java-Klassenbibliotheken ist folgender Konstruktor:

```
GridBagConstraints(int gridx, int gridy, int gridwidth, int
gridheight, double weightx, double weighty, int anchor, int
fill, Insets insets, int ipadx, int ipady)
```

In der eigentlichen Anwendung sieht das dann so aus:

```
new GridBagConstraint(5, 10, 1, 1, 1.0, 2.0, CENTER, BOTH,
emptyInsets, 1, 1)
```

Selbst bei Kenntnis dieser Klasse ist dieses Konstruktor-Monster kaum verständlich. Ein wenig entschärfen lässt sich das ganze durch Kommentare und Formatierung (aber wehe, wenn die IDE den Code umformatiert!):

```
new GridBagConstraint(
    5, 10, // grid position
    1, 1, // cell size
    1.0, 2.0, // weight
    CENTER, // anchor
    BOTH, // fill
    emptyInsets,
    1, 1) padding
```

Die einzig sinnvolle Chance, dieses Monster zu bändigen, ist das Auslagern der Argumente in ein Argument-Objekt, das einzelne Setzen der Felder des leer erzeugten Objektes oder die Verwendung eines Patterns.

Setter (bzw. Direktzugriffe auf die Attribute, der `GridBagConstraint` ist in mehrerer Hinsicht ein Negativbeispiel) blähen unseren Client-Code deutlich auf, sind aber machbar.

Ein Argument-Objekt entfällt, da die Klasse nicht unter unserer Kontrolle ist.

Bleibt noch das Pattern. Eine Möglichkeit wäre das schon erwähnte BUILDER-Pattern:

```
GBCBuilder.create(10, 2).weight(1.0, 2.0)
.anchor(CENTER).insets(emptyInsets).build()
```

Immer noch ein Monster, aber immerhin verständlich (natürlich haben wir in dem Beispiel von der Möglichkeit Gebrauch gemacht, default-Werte nicht noch einmal zu setzen).

5.4.6 Flags

Sehr schwer zu lesen sind Methoden, deren einziger Parameter eine bool'sche Variable ist (mit Ausnahme von Settern natürlich). So eine Methode lässt sich kaum so benennen, dass die Bedeutung auf Client-Seite klar ist (in der Signatur mag es ja noch einigermaßen verständlich sein, aber wir wollen ja gerade verhindern, dass der Leser unnötig hin und her springen muss).

Flag-Methoden



- Flag Methoden sind Methoden mit einem bool'schen Argument:

```
public void paint(boolean isSelected)
```

- Mag noch verständlich sein, aber im Aufruf?

```
public void paint(boolean isSelected)
```

- Besser als zwei getrennte Methoden:

```
public void paintAsSelected()
```

```
public void paintAsUnselected()
```

- Oder mit einer Enumeration

```
public void paintAsSelected()
```

```
public void paintAsUnselected()
```

```
...
```

```
paint(SELECTED);
```

```
figure.paint(true)
```

Was soll uns diese Methode suggerieren? In diesem konkreten Fall lautete die Signatur dazu:

```
public void paint(boolean isSelected)
```

Was zumindest schon besser verständlich ist, aber eben nur, wenn der Leser zwischen den Klassen hin und her wechselt. Besser wäre es gewesen, einfach zwei Methoden zu schreiben:

```
public void paintAsSelected()
```

```
public void paintAsUnselected()
```

Die Unterscheidung, welche davon aufgerufen werden soll, wäre jetzt in den Client-Code ausgelagert worden (der damit drei Zeilen länger, aber dafür verständlicher geworden wäre).


Außerdem werden die beiden Methoden wahrscheinlich ein unterschiedliches Verhalten an den Tag legen, also würde unsere Methode wahrscheinlich sogar mehr als eine Sache machen.

Das gleiche Problem kann natürlich auch bei Methoden mit mehr als einem Parameter auftauchen.

Regel 5-10: Flag-Methoden sollten vermieden werden, besonders bei Monaden. (Lesb)

5.4.7 Ausgabe Parameter

Eine weitere Klasse von Methoden, die konzeptionell nur schwer zu erfassen sind, sind Methoden, die ihre Argumente verändern (sogenannte Output-Argumente). Bei traditionellen Funktionen werden die Eingabeparameter nicht angefasst, sondern ein Ergebnis als Rückgabewert zurückgeliefert. Weichen wir von diesem Grundsatz ab, so wird die Methode schwerer zu erfassen.

Ausgabe Parameter		
<ul style="list-style-type: none">▪ Ausgabe Parameter, also Parameter, die durch die Methode verändert werden, verschlechtern das Verständnis deutlich <pre>... DataBasket basket = ... basket.addToList(orders); ...</pre> <ul style="list-style-type: none">▪ Frage: Wer schreibt hier in welche Liste?▪ Deutlicher wird es, wenn der Ausgabe-Parameter auch noch (überflüssigerweise) zurückgegeben wird: <pre>... DataBasket basket = ... orders = basket.addToList(orders); ...</pre> <ul style="list-style-type: none">▪ Das geht natürlich nicht, wenn mehr als ein Ausgabe-Parameter genutzt wird		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	5 Seite 27

Schauen wir uns folgendes Beispiel an:

```
...
DataBasket basket = ...
Basket.addToList(orders);
...
```

Wo steckt hier das Problem? Die Frage ist, wer schreibt hier in welche Datenstruktur. In dem obigen Beispiel schreibt der Datenkorb seine vollständigen Bestellungen in eine Liste namens `orders`. Es könnte aber auch genau umgekehrt sein, d.h. die Liste `orders` wird dem Warenkorb hinzugefügt.

Man hätte das Ganze entschärfen können, indem die Methode `addToList` die modifizierte Liste noch einmal zurückgeliefert hätte:

```
...
DataBasket basket = ...
orders = Basket.addToList(orders);
...
```

Die Zuweisung ist zwar codetechnisch völlig überflüssig, verbessert die Lesbarkeit ungemein. Kompliziert wird es allerdings, wenn wir mehr als ein Output-Argument haben. Das motiviert uns zu einer weiteren Regel:

Regel 5-11: Methoden sollten höchstens ein Output-Argument besitzen, dieses sollte als Rückgabewert gedoppelt werden. (Lesb)

5.4.8 Argument-Objekte

Argument Objekte



```
public Rectangle(int left, int top, int right, int bottom);
public Rectangle(Point upperLeft, Point lowerRight);
```

- und im Aufruf:

```
new Rectangle(0, 0, 10, 10)
new Rectangle(new Point(0, 0), new Point(10, 10))
```

- Werden die Punkte an einer anderen Stelle als direkt im Aufruf erstellt, so ist der Aufruf auch wieder deutlich kompakter und lesbarer

```
new Rectangle(origin, lowerRight)
```

Betrachten wir zwei Konstruktoren:

```
public Rectangle(int left, int top, int right, int bottom);  
public Rectangle(Point upperLeft, Point lowerRight);
```

Welche von beiden Varianten ist besser zu lesen? Die zweite hat zumindest weniger Argumente, was ja laut den oberen Absätzen die bessere Lösung ist. Aber stimmt das wirklich? Betrachten wir beide Varianten als Aufruf:

```
new Rectangle(0, 0, 10, 10)  
new Rectangle(new Point(0, 0), new Point(10, 10))
```

Haben wir wirklich etwas gewonnen (außer Schreibarbeit)? Natürlich haben wir das, denn jetzt passen die Abstraktionsebene der Methode und die Abstraktionsebene der Argumente wieder zusammen. Ein Rechteck besteht eben konzeptionell nicht aus vier Koordinaten, sondern aus zwei Punkten, selbst wenn es technisch einfach vier Koordinaten-Felder hat.¹¹

Werden die Punkte an einer anderen Stelle als direkt im Aufruf erstellt, so ist der Aufruf auch wieder deutlich kompakter und lesbarer.

```
new Rectangle(origin, lowerRight)
```


Aus Lesbarkeitsgründen ist es also sinnvoll, zusammengehörige Argumente in einem Transfer-Objekt zusammenzufassen.

Leider hat diese Technik einen gravierenden Nachteil. Es werden Unmengen an Wegwerf-Objekten erstellt, die das System deutlich ausbremsen können.

¹¹ Aber das wären natürlich wieder Implementierungsdetails, die der Client ja gar nicht sehen dürfte.

5.5 Stil

In diesem letzten Abschnitt wollen wir uns noch ein wenig näher mit gutem Stil für Methoden beschäftigen. Einige Punkte haben wir dazu schon angesprochen oder impliziert, hier wollen wir das Ganze zu einem sauberen Abschluss bringen.

Stil	
<ul style="list-style-type: none"> ▪ Seiteneffekte <ul style="list-style-type: none"> ▪ sind Effekte einer Methode, die den Zustand verändern oder Aktionen im System auslösen ▪ Aus dem Methodennamen sollte immer klar hervorgehen, ob eine Methode Seiteneffekte hat oder nicht ▪ Command Query Separation <ul style="list-style-type: none"> ▪ Eine Methode sollte entweder etwas zurückliefern oder etwas tun (verändern) ▪ Schlechtes Beispiel: <pre>if (knownNames.add("Peter")) ... if (positions.set("teamleader", "Peter")) ...</pre> ▪ Mehrere Exit Punkte <ul style="list-style-type: none"> ▪ sind bei kurzen Methoden vollkommen akzeptabel ▪ Können durch die IDE hervorgehoben werden ▪ Rekursionen <ul style="list-style-type: none"> ▪ sind kurz und elegant ▪ aber nur mit Aufwand nachzuvollziehen 	
<div style="display: flex; justify-content: space-between; font-size: small;"> 1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH Clean Code - Professionelle Codeerstellung und Wartung 5 Seite 30 </div>	

5.5.1 Seiteneffekte

Unter einem Seiten-Effekt verstehen wir die Veränderung eines Zustandes durch die Methode. Typische Seiteneffekte sind das Setzen von Feldern oder das Ausgeben auf der Konsole.

Eine Seiteneffekt-freie Methode verändert dementsprechend den inneren Zustand unseres Systems nicht. Das bedeutet, wenn wir nebenläufige Zugriffe außen vor lassen, dass das wiederholte Aufrufen einer Methode mit denselben Argumenten auch immer zum selben Ergebnis führt.

Besonders gefährlich sind Seiteneffekte, wenn aus dem Namen der Methode nicht hervorgeht, dass ein Seiteneffekt eintritt.

5.5.2 Befehl oder Abfrage (Command Query Separation)

Jede Methode sollte entweder eine Abfrage oder ein Befehl sein, d.h. sie sollte entweder etwas tun, oder etwas zurückliefern, aber nicht beides. Alles andere führt dazu, dass ein Entwickler mit der genutzten API vertraut sein muss, um den aufrufenden Code zu verstehen. Betrachten wir folgenden Aufruf.

```
if (knownNames.add("Peter")) ...
```

Was sagt die Antwort aus? Hier könnte man noch (richtig) vermuten, dass der Aufruf der `add()`-Methode `true` zurückliefert, wenn das Hinzufügen erfolgreich war, in diesem Fall also, wenn `knownNames` noch keinen Eintrag namens „Peter“ besessen hat.

Wie sieht es mit folgender Methode aus?

```
if (positions.set("teamleader", "Peter")) ...
```

Bedeutet eine positive Antwort, dass der momentane Teamleader Peter heißt? Oder dass der Teamleader erfolgreich auf Peter gesetzt wurde (weil es tatsächlich eine Position namens „teamleader“ gibt)? Wir können zwar versuchen, die Lesbarkeit durch einen besseren Namen zu verbessern (`checkIfExistsAndSet`), aber das hilft auch nicht viel.

Die einzig sinnvolle Lösung, die uns bleibt, ist den Befehl (setze den Wert) von der Abfrage zu trennen:

```
if (positions.entryExists("teamleader")) {  
    positions.set("teamleader", "Peter");  
    ...  
}
```

Regel 5-12: Methoden sollten wenn möglich *entweder* Abfragen *oder* Befehle sein. (Lesb)

Allerdings gibt es natürlich gute Gründe, gegen diese Regel zu verstoßen, insbesondere in der nebenläufigen Programmierung müssen wir uns abstützen, dass Methoden sowohl Abfragen als auch Befehle sind. Das sollte dann aber aus dem Namen zweifelsfrei hervorgehen.

5.5.3 Mehrere Exit-Punkte

Ein alter Programmiergrundsatz lautet, dass jede Routine (also auch jede Methode) nur genau einen Eingang und einen Ausgang haben soll. Wir werden diesen Grundsatz allerdings nicht anwenden. Dafür gibt es zwei Gründe:

- Unsere Methoden sollen schließlich kurz gehalten werden. Das künstliche Zusammenfassen von Exit-Punkten resultiert aber in der Regel in einer zusätzlichen lokalen Variable, und damit in mehr Code
- Das Argument der Unübersichtlichkeit ist nicht wirklich schwerwiegend, wenn unsere Methoden nur wenige Zeilen lang sind, zumal moderne IDEs in der Lage sind, uns Exit-Punkte unseres Codes gesondert zu markieren.

Wir werden also sehr wohl mehr als einen Exit-Punkt verwenden, wenn sich das anbietet (belegt durch einen vernünftigen „To“-Satz natürlich).

5.5.4 Rekursionen

Eine rekursive Funktion (nur für Funktionen macht das überhaupt Sinn) ist eine Funktion, die sich selber aufruft (entweder direkt, oder über eine oder mehrere andere Funktionen hinweg. Das Ziel dabei ist, dass ein Problem in Teile zerlegt wird, und diese Teile dann wiederum durch die gleiche Funktion behandelt werden.

Auf diese Art und Weise sind relativ elegante Lösungen möglich, die insbesondere in der funktionalen Programmierung genutzt werden. Die Formulierung einer rekursiven Funktion ist damit dicht an einer mathematischen Funktion.

Eine rekursive Funktion besteht aus zwei Bestandteilen: den Abbruchbedingungen (auch Spezialfall genannt) und dem rekursiven Anteil (auch allgemeiner Teil genannt).

Ein Beispiel:

Die Summe einer Liste von Zahlen wird folgendermaßen berechnet:

- *Für eine leere Liste ist sie 0*
- *Für eine nicht-leere Liste ist sie das erste Element + die Summe der Restliste*

Im Code sieht das folgendermaßen aus (`head()` und `tail()` sind dabei Methoden, die das erste Element, bzw. eine Restliste zurückliefern):

```
public int sum(List elements) {  
    if (elements.isEmpty()) return 0;  
    return elements.head() + sum(elements.tail());  
}
```

Diese Lösung ist kurz und elegant – und leider ziemlich ineffizient und unnötig. Bevor wir uns aber darüber Gedanken machen, prüfen wir, ob die Methode gegen Regeln verstößt.

Die Methode ist kurz, sie macht nur eine Sache und sie hat nur ein Argument (sie ist also eine Abfrage). Sie hat auch keine Nebeneffekte. Wie sieht es mit der Stepdown-Regel aus?

To sum all elements, either return 0, if there is no element or add the head element to the sum of the tail.

Das sieht zumindest nicht verkehrt aus. Das einzige, was man anmerken könnte wäre, dass wir natürlich nicht bei jedem Aufruf eine Abstraktionsebene tiefer gehen. Das ist allerdings nicht weiter dramatisch, zumindest gehen wir die Ebenen nicht wieder hinauf.

Wo liegt jetzt der Nachteil? Zunächst erfordern Rekursionen immer einen Moment des Nachdenkens (wenn man nicht gerade Vollzeit-Mathematiker ist), und genau das wollen wir ja eigentlich vermeiden.

Weiterhin gibt es ja genau für das Problem (der Summe der Elemente einer Liste) ja eine sehr einfache Möglichkeit: das Durchiterieren der Liste und Aufsummieren der Werte.

Drittens ist eine Rekursion ziemlich Ressourcen-hungrig, wenn der Compiler dafür nicht spezielle Unterstützung bietet.

Für einfache Beispiele (wie sie leider oft in Informatik-Büchern stehen) ist Rekursion also eher nicht geeignet – das gilt insbesondere für die beiden Standardbeispiele, Fibonacci-Zahlen und Fakultät!

Trotzdem hat Rekursion natürlich ihre Anwendungsgebiete: Fachcode, der von Mathematikern gepflegt wird und die fachlichen Berechnungen der Software implementiert, oder wenn die Anforderungen bereits rekursiv formuliert sind.

Es geht also nicht darum, Rekursionen absolut zu vermeiden, sondern darum, sie nicht einfach der Eleganz wegen zu verwenden. Was wir aber unbedingt vermeiden sollten, sind zyklische Rekursionen (A ruft B auf und B wieder A). Diese sind schwer nachzuvollziehen und verstoßen auch mit ziemlicher Sicherheit gegen die Stepdown-Regel.

5.6 Zusammenfassung

In diesem Kapitel haben wir eine Reihe von Regeln und Verfahren besprochen, die im ersten Moment äußerst extrem erscheinen. Insbesondere das Reduzieren unserer Methoden auf eine Handvoll Zeilen erfordert ein deutliches Umdenken.

Der Aufwand lohnt sich allerdings. Indem wir unsere Methoden auf diese Art und Weise zusammenbauen, lässt sich unser Code wie eine Geschichte lesen.

Natürlich sind unsere Methoden im ersten Wurf in der Regel nicht so strukturiert und sauber wie in diesem Kapitel gefordert. Zunächst implementieren wir unsere Funktionalität. Aber danach sollten wir uns eben die Zeit nehmen, unseren Code lesbar zu machen.

6

Kommentare und Dokumentation

6.1	Einleitung.....	6-3
6.1.1	Lesbarer Code	6-4
6.2	Gute Kommentare	6-6
6.2.1	Rechtliche Hinweise.....	6-7
6.2.2	Klarstellungen	6-8
6.2.3	Absichtserklärungen	6-8
6.2.4	Design Patterns	6-8
6.2.5	Regelverstöße	6-9
6.2.6	Unterstreichungen.....	6-9
6.2.7	Formale Kommentare	6-9
6.3	Schlechte Kommentare	6-12
6.3.1	Unverständliche Kommentare.....	6-12
6.3.2	Redundanzen	6-13
6.3.3	Forcierte Kommentare	6-13
6.3.4	Codehistorien.....	6-14
6.3.5	Klammer-Kommentare.....	6-15
6.3.6	Auskommentierter Code	6-16
6.3.7	Informationsüberfluss.....	6-16
6.3.8	TODOs.....	6-17
6.3.9	Nicht-öffentliche formale Kommentare.....	6-18
6.4	Testfälle als Dokumentation	6-19

6.5	Zusammenfassung	6-20
-----	-----------------------	------

6 Kommentare und Dokumentation

6.1 Einleitung

In diesem Kapitel werden wir uns mit Kommentaren und Dokumentation beschäftigen. Mit Dokumentation meinen wir dabei formale Kommentare im Quellcode, aus denen später eine API-Beschreibung generiert werden kann (mit Javadoc oder Doxygen). Nutzerdokumentation o.Ä. ist damit natürlich nicht gemeint!

Eine Erkenntnis die das Kapitel hoffentlich zeigen wird, ist das insbesondere Kommentare (nicht Dokumentation) häufig nicht nur unnötig, sondern kontraproduktiv sind.

Warum wird hier so vehement gegen Kommentare argumentiert? Weil Kommentare ein großes Problem mit sich bringen: Sie werden in der Regel nicht weitergepflegt. Das bedeutet, dass ein Kommentar zum Zeitpunkt, an dem er geschrieben wurde sinnvoll gewesen sein mag, aber wurde der kommentierte Code seitdem geändert, ist nicht sicher gestellt, dass auch der Kommentar angepasst wurde.

Ein Beispiel: Ein weiteres Problem von Kommentaren ist ihre mangelnde Überprüfbarkeit. Alles, was nicht automatisiert geprüft werden kann (i.d.R. im Build-Prozess), muss entweder manuell geprüft werden, oder verworfen!

Das Problem mit Kommentaren



```
/**
 * Add a new Action to the manager. Returns true if the
 * action
 * is already existant. If the action is already
 * registered,
 * it is NOT replaced.
 * @param action the action to add
 * @return True if an action with the same name has
 * already been added, false otherwise.
 */
public void addAction(JaspiraAction action)
```

- Kommentare veralten schnell und werden nicht immer angepasst.

6.1.1 Lesbarer Code

Kommentare sind deshalb hoffentlich selten notwendig, weil uns ja unser bisheriges Bemühen, insbesondere die beiden letzten Kapitel an eine Stelle gebracht haben sollte, an denen unser Code so gut verständlich ist, dass wir eigentlich keine Kommentare brauchen.

Lesbarer Code



- Besser als Kommentare ist es, den Code lesbar zu gestalten

```
// is order eligible for free shipping  
if (order.getValue() > FREE_SHIPPING_LIMIT  
    || isPremiumMember(order.getCustomer()))
```

- Oder besser so:

```
if (isEligibleForFreeShipping(order))
```

- Dann kann das folgende nicht passieren:

```
// is order eligible for free shipping  
Customer customer = order.getCustomer();  
customer.addToOrderHistory(order);
```

```
if (order.getValue() > FREE_SHIPPING_LIMIT  
    || isPremiumMember(order.getCustomer()))
```

Häufig, wenn auch nicht immer, kann der Kommentar durch eine entsprechende Methode oder Variable ersetzt werden. Vergleichen Sie folgende Abfrage

```
// is order eligible for free shipping
if (order.getValue() > FREE_SHIPPING_LIMIT
    || isPremiumMember(order.getCustomer()))
```

mit ihrer Variante ohne Kommentar:

```
if (isEligibleForFreeShipping(order))
```

Abgesehen davon, dass die zweite Variante deutlich kürzer ist, kann die Erklärung dabei auch nie vom eigentlichen Code getrennt werden, wie es hier passiert ist:

```
// is order eligible for free shipping
Customer customer = order.getCustomer();
customer.addToOrderHistory(order);

if (order.getValue() > FREE_SHIPPING_LIMIT
    || isPremiumMember(order.getCustomer()))
```

Und beide Varianten haben natürlich den gleichen Aufwand. Bei der kommentierten Version schreiben wir erst den Kommentar, um auszudrücken, was wir vorhaben (oder überlegen es uns zumindest), bei der zweiten Version schreiben wir das, was wir vorhaben, einfach als Methodenaufruf nieder. Und erst dann erstellen wir die neue Methode (was mit einer modernen IDE wieder nur einige Tastendrücke erfordert).


Wenn wir einen Kommentar (Ausnahmen gibt es natürlich, s.u.) in unserem Code verwenden, sagen wir eigentlich damit aus: Wir sind nicht in der Lage, uns alleine mit unserem Code auszudrücken.

Gerade bei komplexerem Code gibt es eine Tendenz, diesen (mehr oder weniger sinnvoll) zu kommentieren. Das ist aber der falsche Ansatz! Ist der Code zu komplex, sollte er vereinfacht werden. Kommentare sind kein Freischein für schlechten Code.

Regel 6-1: Bevor ein Kommentar gesetzt wird, um Code zu erklären, sollte immer erst versucht werden, den Code selbst verständlicher zu gestalten. (Lesb)

6.2 Gute Kommentare

Natürlich sind nicht alle Kommentare schlecht. Wir werden im Folgenden eine Reihe von Kommentartypen besprechen, die notwendig, sinnvoll oder zumindest unvermeidbar sind.


Gute Kommentare


- **Rechtliche Hinweise**

```
/*
 * (c) 2009, 2010 by Integrata AG, all rights reserved
 * Release under Apache License 1.0
 */
```
- **Klarstellungen**

```
Pattern dateTimePattern = Pattern.compile(
    // 31 . 08 . 2004 16 : 28 +1
    // 1 . 4 . 04 4 : 15
    "\\d{1,2}\\.\\d{1,2}\\.((\\d{4})|(\\d{2})) \\d{1,2}:\\d{2}
    (+\\d)?")
```
- **Absichtserklärungen**
 - Warum wurde dieses Stück Code genau so geschrieben?

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code - Professionelle Codeerstellung und Wartung
6 Seite 6

Gute Kommentare


- **Design Patterns**
 - Wenn der Name des Patterns nicht sowieso Teil des Klassen/Methoden-Namens ist.

```
/**
 * Provides Singleton access to the DataBase Layer.
 * ...
 */
public class DataBaseControl {
```
- **Regelverstöße**
 - Hinweis darauf, dass gegen eine Regel/Konvention verstoßen wurde, welche, und warum
- **Unterstreichungen**
 - Zum Hervorheben eines essentiellen Schrittes im Code, der sonst vielleicht überlesen würde

```
members.clear() // necessary for Garbage Collection to reclaim members
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH
Clean Code - Professionelle Codeerstellung und Wartung
6 Seite 7

6.2.1 Rechtliche Hinweise

Diese Art von Kommentar interessiert uns als Entwickler in der Regel recht wenig. Sie ist formalisiert, sieht für ein Projekt immer gleich aus, wird durch die IDE normalerweise ausgeblendet, und die gleiche IDE legt sie beim Erstellen einer neuen Datei auch mit an.

Trotzdem sind sie natürlich wichtig!

```
/*  
 * (c) 2009, 2010 by Integrata AG, all rights reserved  
 * Release under Apache License 1.0  
 */
```

Das einzige, worauf wir hier achten sollten ist, dass ein derartiger Kommentar eben nicht einen vollständigen Lizenztext beinhalten sollte, sondern in der Regel nur den Namen und die Version der Lizenz aufzählt (wenn es sich um einen Standard-Lizenz handelt) oder auf ein externes Dokument verweist.

6.2.2 Klarstellungen

Klarstellungen dienen dazu, einen Code, der inherent unleserlich oder schwer verständlich ist (was nicht immer zu vermeiden ist, gerade, wenn externe Bibliotheken genutzt werden). Ein häufiges Beispiel dafür sind Reguläre Ausdrücke:

```
Pattern dateTimePattern = Pattern.compile(  
    // 31 . 08 . 2004          16 : 28 +1  
    // 1 . 4 . 04             4 : 15  
    "\\d{1,2}\\.\\d{1,2}\\.((\\d{4})|(\\d{2}) \\d{1,2}:\\d{2} (+\\d)?)"
```

6.2.3 Absichtserklärungen

Eine Absichtserklärung dient dazu, deutlich zu machen, was die Intention des Programmierers war, um ein bestimmtes Stück Code genau so zu schreiben.

```
@Override  
public boolean equals(Object other) {  
    if (other == null) return false;  
    if (this == other) return true;  
  
    if (other.getClass() == this.getClass()) {  
        //...  
    }  
    // Value Objects can be equal to actual objects  
    else if (other.getClass() == PersonValue.class) {  
        return toValueObject().equals(other);  
    }  
}
```

6.2.4 Design Patterns

Wird in einem Stück Code ein Design-Pattern umgesetzt und ist das nicht sofort ersichtlich, sollte in einem Kommentar darauf hingewiesen werden. Besser ist allerdings, die Namen der beteiligten Klassen entsprechend zu wählen.

6.2.5 Regelverstöße

Aus Sicht des Autors die wichtigste Art von Kommentar. In Programmen wird es immer wieder notwendig sein, gegen Konventionen, Regel oder sogar Contracts zu verstoßen. Derartige Verstöße müssen unbedingt kommentiert werden¹, sonst könnte es passieren, dass ein anderer Programmierer den vermeintlichen Fehler korrigiert.

Regel 6-2: Regelverstöße müssen durch einen Kommentar markiert und begründet werden. (Lesb, Vers)


6.2.6 Unterstreichungen

Unterstreichungen sind Hervorhebungen von Code, der sonst überlesen würde. Also ein eigentlich trivialer Schritt, der hier aber eine besondere Bedeutung hat.

6.2.7 Formale Kommentare

Formale Kommentare sind das, was wir eingangs als Dokumentation bezeichnet haben. Es sind Kommentare, aus denen später eine API-Dokumentation generiert wird.

Gute Kommentare



- Formale Kommentare
 - JavaDoc, Doxygen oder POD, also formale Formate aus denen automatisiert eine Dokumentation erstellt werden kann
 - JavaDoc ist mit seiner HTML-Syntax allerdings ein schlechtes Beispiel:

```
/**
 * Returns an XML-Representation of this MemberList.
 * Generated Code has the following format:
 *
 * <members><br/>
 *   <person><br/>
 *     <firstname>Dieter</firstname><br/>
 *     <lastname>Maier</lastname><br/>
 *     <birthday>1973-15-21</birthday><br/>
 *   </person><br/>
 * </members><br/>
 */
public void toXML() {
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

6 Seite 8

¹ Und zwar mit einer Begründung!

Diese Kommentare für die öffentliche² API (und nur für diese) können ausführlich und detailliert die Benutzung der Routinen beschreiben und sind damit eine wertvolle Hilfe für jeden Programmierer, der auf unsere Module zugreifen will.

Regel 6-3: Formale Kommentare sollten dem Visions-Prinzip folgen. (Lesb, Vers)

Das bedeutet in diesem Fall, dass der erste Satz der Klassen oder Methoden-Beschreibung für sich alleine verständlich sein sollte. Der restliche Kommentar dient zur Klarstellung, für Details und ggf. für Anwendungsbeispiele.

Der erste Satz ist der, der auch in Übersichten in der generierten Dokumentation auftaucht (der erste Satz der Klassen in der Beschreibung des Pakets, der erste Satz der Methoden in der Beschreibung der Klasse).

Regel 6-4: Formale Kommentare sollten im *Quellcode* lesbar sein. (Lesb)

Die meiste Zeit über wird ein Entwickler nicht auf die generierte Dokumentation, sondern direkt in den Quelltext schauen. Deshalb ist es ungemein wichtig, dass die Dokumentation auch im Quelltext verständlich ist.

² Die öffentliche API setzt sich dabei aus *public* Elementen und *protected* Elementen zusammen, letztere aber nur, wenn die Klasse zur Vererbung gedacht ist.


Leider verwenden viele Systeme HTML, um ihre formalen Kommentare zu formatieren, mit dem Ergebnis, dass der Kommentar im schlimmsten Fall im Sourcecode unverständlich ist:

```
/**
 * Returns an XML-Representation of this MemberList.
 * Generated Code has the following format:
 *
 * <members><br/>
 *  <person><br/>
 *   <firstname>Dieter</firstname><br/>
 *    <lastname>Maier</lastname><br/>
 *    <birthday>1973-15-21</birthday><br/>
 *  </person><br/>
 * </members><br/>
 */
public void toXML() {
```

6.3 Schlechte Kommentare

Neben guten Kommentaren gibt es natürlich eine ganze Reihe von schlechten Kommentaren, also Kommentare, die nichts zur Lesbarkeit beitragen oder diese sogar noch verschlechtern. Die häufigsten wollen wir hier aufzählen.

Schlechte Kommentare



- Unverständliche Kommentare
 - Das kann z.B. Sinn-entstellender Satzbau oder ein halbfertiger Satz sein
- Redundanzen

```
// Register implementations in the service registry
initServices(services);

// Initialize the persistence layer
initPersistence();

// Reads all models
initModels();

// Initializes advanced of the system
initServices(services2);

// Register a shutdown hook that allows correct database shutdown
registerShutdownHook();

// Initializes the remote services (if necessary).
initRemoting();
```

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung6 Seite 11

6.3.1 Unverständliche Kommentare

Unverständliche Kommentare sind Kommentare, die in der Hitze des Gefechts unlesbar herausgekommen sind. Gründe dafür mögen die Uhrzeit oder Termindruck sein.

Fest steht aber, dass ein unverständlicher Kommentar schlechter als gar kein Kommentar ist.

6.3.2 Redundanzen

Redundante Kommentare sind Kommentare, die eigentlich nur wiederholen, was bereits im Code steht. Darunter fallen zum Einen Aufrufe, zum Anderen aber auch formale Kommentare:

```
// Register implementations in the service registry
initServices(services);

// Initialize the persistence layer
initPersistence();

// Reads all models
initModels();

// Initializes advanced of the system
initServices(services2);

// Register a shutdown hook that allows correct database shutdown
registerShutdownHook();

// Initializes the remote services (if necessary).
initRemoting();

/**
 * Initializes the persistence layer.
 */
protected void initPersistence()

/**
 * Shuts down the persistence layer.
 */
protected void shutdownPersistence()
```

6.3.3 Forcierte Kommentare

Diese Kommentare gehen in eine ähnliche Richtung. Häufig findet man in den internen Programmierrichtlinien (so diese denn existieren) die Forderung, jede öffentliche Methode mit einem Kommentar zu versehen.

Schlechte Kommentare



- Forcierte Kommentare
 - Kommentare, die nur geschrieben werden, weil eine Richtlinie es verlangt („Jede Methode muss kommentiert sein“)
 - Pflicht, Getter und Setter zu kommentieren
- Codehistorien


```
/*
 * Demo.java
 *
 * 18.03.03 sp Initial Version
 * 15.04.03 sp added Lifecyclemethods
 * 18.04.03 jf implemented Comparable
 * 30.05.03 sp general Refactoring
 */
```

 - Dafür gibt es die Sourcecode Verwaltung / unser Ticket-System

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

6 Seite 12

Der Effekt davon ist, dass die Programmierer Zeit damit verbringen, den sinnvollen Namen, den sie ihrer Methode gegeben haben im Kommentar zu doppeln oder mit einer alternativen Beschreibung, die exakt das gleiche aussagt zu versehen (siehe `initPersistence()` und `shutdownPersistence()` im obigen Beispiel)

Ein gutes Beispiel sind Getter und Setter. Diese mit Kommentaren zu versehen, ist schlicht und ergreifend Code-Müll!

6.3.4 Codehistorien

In einigen (gerade älteren) Projekten findet sich im Kopf, seltener am Ende einer Datei eine Beschreibung, wer was wann geändert hat, also eine Historie der Datei.

```
/*
 * Demo.java
 *
 * 18.03.03 sp Initial Version
 * 15.04.03 sp added Lifecyclemethods
 * 18.04.03 jf implemented Comparable
 * 30.05.03 sp general Refactoring
 */
```


Das ist bisher nur die harmlose Fassung, die nur größere Änderungen aufzählt. Gelegentlich sieht man aber auch die Variante, dass **jede** Änderung dort verzeichnet sein soll. Mit dem Ergebnis, dass diese Kommentare über mehrere Bildschirmseiten gehen.

Welchen Nutzen haben diese Kommentare (außer Platz zu verschwenden)? Unsere IDE blendet sie normalerweise sowieso aus.

Benötigen wir die Informationen über die Änderungen, so bekommen wir diese auch ohne weiteres von unserer Sourcecode-Verwaltung und/oder unserem Ticketsystem.

6.3.5 Klammer-Kommentare

Schlechte Kommentare



- Klammer-Kommentare

```
for (int i = 0; i < max; i++) {
    try {
        ...
    } // try
    catch (IOException e) {
        ...
    } // catch
} // for
```
- Auskommentierter Code
 - Falls es tatsächlich nötig ist: mit Begründung
- Unnötige Information
 - Eine historische Information zu einem Algorithmus ist fehl am Platz

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung6 Seite 13

Eine weitere Praxis, die früher recht geläufig war, ist das Markieren von schließenden Klammern:

```
for (int i = 0; i < max; i++) {
    try {
        ...
    } // try
    catch (IOException e) {
        ...
    } // catch
} // for
```


Diese Technik hätte nur dann Sinn, wenn unsere Methoden deutlich größer und verschachtelter wären, als wir erreichen wollen.

6.3.6 Auskommentierter Code

Auskommentierter Code hat die Tendenz, der langlebigste Teil unseres Codes zu werden. Keiner traut sich daran, ihn zu löschen, keiner weiß mehr, warum er auskommentiert wurde. Betrachten wir folgendes Beispiel aus der JCommons-Library:

```
this.bytePos = writeBytes(pngIdBytes, 0);  
//hdrPos = bytePos;  
writeHeader();  
writeResolution();  
//dataPos = bytePos;  
if (writeImageData()) {  
    writeEnd();  
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);  
}  
else {  
    this.pngBytes = null;  
}  
return this.pngBytes;
```

Warum wurden diese Zeilen auskommentiert? Wurde vielleicht vergessen, sie wieder ein zu kommentieren?


Wenn schon Code auskommentiert wird, dann sollte auch dabeistehen, warum. Aber besser ist es, ganz darauf zu verzichten (natürlich spricht überhaupt nichts dagegen, für einen Testlauf Code aus zu kommentieren – aber dieser Code darf dann natürlich niemals wieder eingecheckt werden!)

6.3.7 Informationsüberfluss

Information, die nichts mit dem Code zu tun hat, gehört auch nicht in den Code. Von wem ein Algorithmus entwickelt wurde und wie er sich im Laufe der Zeit gewandelt hat, ist für denjenigen, der den Code lesen soll, unerheblich.³

³ Allenfalls eine URL mit weiterführenden Informationen ist hinnehmbar

6.3.8 TODOs


<h4>Schlechte Kommentare</h4> <ul style="list-style-type: none">▪ TODOs<ul style="list-style-type: none">▪ TODOs sind Kommentare, mit denen ein Programmierer kenntlich macht, dass hier noch etwas zu tun ist▪ Sie sind per se nicht schlimm, wenn sie zügig beseitigt werden▪ Die Realität zeigt aber, dass diese Kommentare selten je wieder angefasst werden▪ Sie sollten nicht als „zweites Ticketsystem“ genutzt werden.▪ Nicht öffentliche, formale Kommentare<ul style="list-style-type: none">▪ Private Methode mit formaler JavaDoc zu versehen kostet Zeit und Platz
<div>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung6 Seite 14</div>

Kommentare, die man häufig in Code findet, sind TODO Kommentare. Dabei handelt es sich um Markierungen, mit denen ein Programmierer deutlich macht, dass an dieser Stelle noch etwas getan werden muss, aber aus irgendeinem Grund noch nicht getan werden kann.

Der Vorteil von TODO-Kommentaren ist, dass moderne IDEs in der Lage sind, alle TODOs aus einem Projekt übersichtlich zu präsentieren.

Der gravierende Nachteil ist aber, dass dabei die Gefahr besteht, ein „zweites Ticket-System“ neben dem eigentlichen Projekt-System aufzustellen. Notwendige Arbeitsschritte am Code sollten alle an **einer** Stelle zusammengefasst sein.

Kann man die Sourcecode-Verwaltung so konfigurieren, dass das Einchecken eines neuen TODO-Kommentares automatisch ein Ticket anlegt, kann man dieses Problem aber elegant umgehen.

6.3.9 Nicht-öffentliche formale Kommentare


Code aus dem keine API-Dokumentation erzeugt wird (also nicht-öffentlicher Code) benötigt auch keine formalen Kommentare. Natürlich kann und wird auch dieser Code dokumentiert werden, aber eben nicht formal.

Wo liegt der Unterschied? Formale Kommentare drängen uns noch deutlich mehr formal Zwänge auf, die aber für den Quellcode unerheblich sind. D.h. wir verbrauchen Zeit und Platz für Formalismen, die dazu dienen eine Dokumentation generieren zu können, die niemals gebraucht wird.

Regel 6-5: Nur die öffentliche API sollte formal beschrieben werden. (Lesb)

6.4 Testfälle als Dokumentation

Eine sinnvolle Form der Dokumentation sind Testfälle (siehe dazu auch das Kapitel über Tests). Um die Benutzung einer API zu verstehen, bietet es sich an, ein großes Augenmerk auf die Testfälle zu werfen, denn diese müssen zwangsläufig angepasst werden, wenn der Code verändert wird. Bei Kommentaren kann das ja versäumt werden.

Testfälle als Dokumentation	
<ul style="list-style-type: none">▪ Eine Alternative zur herkömmlichen Dokumentation sind Testfälle▪ Sie zeigen eine API in Benutzung und können somit gleichzeitig als Tutorial dienen▪ Im Gegensatz zur normalen Dokumentation werden sie regelmäßig auf Korrektheit geprüft.	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung
6 Seite 15	

Testfälle sind damit in der Regel aktueller als Kommentare. Es ist eine gute Praxis, Testfälle auch mit diesem Hintergrund zu schreiben. Eine Reihe guter Testfälle erspart damit auch Beispiele und zusätzliche Erklärungen in den Kommentaren.

Noch besser wäre es, wenn die Testfälle selbst Teil der Dokumentation wären, damit hätten wir ein wichtiges Problem gelöst: die Testbarkeit unserer Dokumentation.⁴

⁴ Ein interessantes Projekt unter Java, das dieses Ziel verfolgt sind die Java-Eunnotations (<http://www.eucodos.de/eunnotations/doctract>)

6.5 Zusammenfassung

Wir haben uns in diesem Kapitel mit Grundsätzen für gute Kommentare auseinandergesetzt. Die wichtigsten Punkte dabei waren:

Fazit



- Kommentare sind keine Rechtfertigung für schlechten Code
- Ein Kommentar, der nur geschrieben wird, weil das die Konvention verlangt, ist unnötig – hier sollte die Konvention geändert werden
- Bei Kommentare gilt ganz klar: weniger ist mehr.

7

Code-Formatierung

7.1	Einleitung.....	7-3
7.2	Warum Formatierung.....	7-3
7.2.1	Automatisierte Formatierung.....	7-4
7.2.2	Sourcecode als Kommunikation	7-6
7.3	Die Zeitungsmetapher	7-7
7.3.1	Schlagzeile	7-8
7.3.2	Untertitel	7-8
7.3.3	Der Einstieg / Lead	7-9
7.3.4	Absätze.....	7-9
7.3.5	Reihenfolgen.....	7-12
7.3.6	Die Rubrik	7-13
7.4	Weitere Formatierungsregeln	7-14
7.4.1	Breite und Höhe.....	7-14
7.4.2	Einrückungen.....	7-15
7.4.3	Ausnahmen.....	7-16
7.5	Team Rules!	7-17
7.6	Zusammenfassung	7-18

7 Code-Formatierung


7.1 Einleitung

In diesem (recht kurzen) Kapitel werden wir den Überblick über formale Code-Richtlinien abschließen. Wir werden auf einige Grundsätze eingehen, wie auf die Frage, warum Formatierung heutzutage überhaupt noch ein Thema ist, und einige gute Ansätze für einen sauberen Code betrachten.

Wir werden uns in diesem Kapitel weniger damit beschäftigen, wo Klammern und wo Leerzeichen hinsollen oder nicht, sondern uns mehr auf das große Bild konzentrieren.

7.2 Warum Formatierung

Warum der Code formatiert werden sollte, dürfte hoffentlich klar sein. Die Frage ist, warum müssen wir uns im Zeitalter automatische Code-Formatierung damit auseinandersetzen?

Warum Formatierung?	
<ul style="list-style-type: none">▪ Code wirkt „aus einem Guss“▪ Es ist leichter, sich in Code zurecht zu finden, der in gewohnter Art formatiert ist▪ Bei unterschiedlichen Formatierung (besonders bei automatischer) unter den Entwicklern kann ein Commit in die Sourcecode-Verwaltung unzählige Änderungen anzeigen, die keine sind▪ Sourcecode ist Kommunikation!	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 7 Seite 2

Nun, zum Einen decken Formatieren nicht alles ab, was wir hier besprechen, zum Anderen, muss ja irgendjemand auch den Formatter konfigurieren.

7.2.1 Automatisierte Formatierung

Gerade mit automatischer Formatierung ist es umso wichtiger, dass alle Entwickler die gleichen Formatierungen verwenden. Nicht nur, wie immer als erstes genannt wird, damit der Code aus einem Guss wirkt (was trotzdem wichtig ist), sondern vor allem auch wegen der Zusammenarbeit mit der Sourcecode-Verwaltung.

Die meisten Sourcecode-Verwaltungen arbeiten zeilenorientiert, d.h. bei einem eincheck-Vorgang wird unterschieden zwischen Zeilen, in denen sich etwas geändert hat und Zeilen, die gleichgeblieben sind.

Betrachten wir folgendes Beispiel:

Entwickler A verwendet die Einstellung, dass die öffnenden, geschweiften Klammern in der Zeile der Anweisung stehen sollen (der sogenannte K&R-Stil¹), Entwickler B die Einstellung, dass sie immer in einer eigenen Zeile stehen müssen.

B checkt nun den folgenden, zuletzt von A veränderten Code aus, fügt die markierte Zeile ein und benennt die Methode um:

```
public class ServiceRegistry {
    private Set<Service> registeredServices;

    void registerAndInitializeService(Service service) {
        if (registeredServices.contains(service)) {
            throw new IllegalStateException();
        }
        registeredServices.add(service);
        service.initialize();
    }

    void doSomething() {
        if (checkSomething()) {
            doSomethingElse();
        }
    }
    //...
}
```

¹ Nach Brian Wilson Kernighan und Dennis Ritchie: Ritchie ist der Schöpfer von C, mit Kernighan zusammen hat er außerdem ein fundamentales Buch zur C-Programmierung geschrieben, in dem eben dieser Klammer-Stil empfohlen wird.

Beim Abspeichern tritt nun die automatische Formatierung in Aktion. Checkt er die geänderte Klasse dann wieder ein, so erkennt die Sourcecode-Verwaltung alle markierten Zeilen als Änderungen:

```
public class ServiceRegistry
{
    private Set<Service> registeredServices;

    void registerService(Service service)
    {
        if (registeredServices.contains(service))
        {
            throw new IllegalStateException();
        }
        registeredServices.add(service);
        service.initialize();
    }

    void doSomething()
    {
        if (checkSomething())
        {
            doSomethingElse();
        }
    }
}
```

Will nun Entwickler A überprüfen, was sich am Code getan hat, so bekommt er von der Sourcecode-Verwaltung die Information, dass sich unter anderem die Methode `doSomething()` geändert hat, die aber tatsächlich von niemandem angerührt wurde.

Schlimmstenfalls gehen die wichtigen Änderungen im Rauschen der geänderten Formatierungen vollständig unter und werden übersehen.

7.2.2 Sourcecode als Kommunikation

Unser Quellcode ist eine Art zu kommunizieren. Mit anderen Programmierern und mit uns selbst. Dementsprechend ist unsere Formatierung die äußere Form unserer Kommunikation.


Was wird wohl eher gelesen und ernstgenommen? Einige hastig hingeschmierte Punkte auf einer Serviette oder die gleichen Punkte sauber gedruckt auf einem Blatt Papier?


Wenn wir unseren Quellcode als Medium betrachten, können wir daraus einige Ansätze für die Formatierung herausholen.

Die Formatierung, Anordnung unsere Methoden und selbst die Platzierung der Leerzeilen ist eine *Information!* Auf diese Informationen sollte sich ein Leser verlassen können, ohne sich in jeder Klasse auf einen neuen Stil einstellen zu müssen.

7.3 Die Zeitungsmetapher

Eine gute Klasse wollen wir im Folgenden mit einem Zeitungsartikel vergleichen. Dieser beginnt mit einer Schlagzeile, gefolgt von einem Untertitel und einer Zusammenfassung. Dann folgt der eigentliche Text vom Allgemeinen zum Speziellen. Auf diese Art und Weise wollen wir unseren Code auch aufbauen.

Die Zeitungsmetapher	 integrata cegos	
Eine Klasse soll aufgebaut sein, wie ein Zeitungsartikel		
<ul style="list-style-type: none">▪ Schlagzeile<ul style="list-style-type: none">▪ Liefert das Thema des Artikels▪ Ist das erste Entscheidungsmerkmal, ob Weiterlesen sich lohnt▪ Entspricht dem <i>Klassennamen</i>▪ Untertitel<ul style="list-style-type: none">▪ Ein einzelner Satz, der den Inhalt genauer präzisiert▪ Entspricht der <i>Klassenvision</i>▪ Der Einstieg / Lead<ul style="list-style-type: none">▪ Bei Zeitungsartikeln der fettgedruckte erste Teil▪ Gibt einen detaillierteren Überblick und fasst den Artikel grob zusammen▪ Entspricht dem formalen Klassenkommentar		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	7 Seite 3

Die Zeitungsmetapher	 integrata cegos	
<ul style="list-style-type: none">▪ Absätze<ul style="list-style-type: none">▪ Eine Reihe von logisch zusammenhängenden Sätzen▪ Sind durch Leerzeilen (ggf. halb) von einander getrennt▪ Entspricht <i>Methoden</i>▪ Reihenfolgen<ul style="list-style-type: none">▪ Die Reihenfolge der Methoden orientiert sich ebenso an einem Zeitungsartikel▪ Vom allgemeinen Überblick geht man langsam ins Detail▪ Das Auge sollte nur von oben nach unten fließen müssen▪ Zusammengehöriges sollte dicht bei einander stehen		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	7 Seite 4

7.3.1 Schlagzeile

Die Schlagzeile, also der Klassenname liefert uns einen prägnanten Überblick über das allgemeine Thema der Klasse. Sie muss aussagekräftig genug sein, dass ein Blick ausreicht, um zu erkennen, ob der Inhalt für den Leser relevant ist, oder nicht.

Insbesondere muss die Schlagzeile sich soweit von anderen Schlagzeilen abheben, dass der Leser diese auf einen Blick unterscheiden kann.

Einen Unterschied zur Schlagzeile in der Zeitung gibt es allerdings. Schlagzeilen sind häufig reißerisch oder provozierend gewählt, um ein erstes Interesse zu wecken. Das wollen wir hier natürlich so nicht umsetzen.

7.3.2 Untertitel

Der Untertitel eines Artikels präzisiert den Inhalt näher – aber immer noch in einem Satz. Er wird dazu genutzt, Informationen unterzubringen, die den Titel überfrachten würden.

Die Entsprechung des Untertitels in unserem Code ist natürlich die Vision. Sie ist der zweite Platz, auf den unser Blick fällt, wenn wir anhand des Klassennamens entschieden haben, dass die Klasse näher betrachtet werden soll.

7.3.3 Der Einstieg / Lead

Als Einstieg oder Lead bezeichnet man in einem Zeitungsartikel den fett gedruckten ersten Teil, der einen Überblick über den Inhalt gibt. Die wichtigsten Stichpunkte werden hier zusammengefasst.

In unserem Code ist der Lead der formale Kommentar unserer Klasse. Er sollte beschreiben, was wir von der Klasse zu erwarten haben und wie wir damit umgehen sollen. Wichtige Punkte im Code selbst werden hier bereits hervorgehoben.

7.3.4 Absätze

Ein Absatz stellt eine Reihe von logisch zusammenhängenden Sätzen dar. In unserem Code: eine Methode! Absätze sind voneinander optisch getrennt – mit einer halben oder einer ganzen Leerzeile.

Diese Praxis sollten wir unbedingt für unseren Code übernehmen:

Regel 7-1: Methoden sollten voneinander durch eine Leerzeile getrennt werden. (Lesb)

Zeigen wir das an einem Beispiel:

```
private void initServices(List<Service> services) {
    for (Service next : services)
        registerAndInitService(service);
}

private void registerAndInitService(Service service) {
    if (service instanceof LifecycleSupport)
        ((LifecycleSupport) next).initialize();
    serviceRegistry.register(o);
}

private void shutdownServices(List<Service> services) {
    for (Service next : services)
        shutdownService(service);
}

private void shutdownService(Service service) {
    if (service instanceof LifecycleSupport)
        ((LifecycleSupport) service).shutdown();
}

protected void initPersistence() {
    PersistenceContextProvider provider = getPersistenceContextProvider()
    if (provider != null)
        provider.initialize();
}

protected void shutdownPersistence() {
    PersistenceContextProvider provider = getPersistenceContextProvider()
    if (provider != null)
        provider.shutdown();
}

private void initRemoting() {
    try {
        doInitRemoting();
    } catch (Exception e) {
        throw new EngineException("Initialization", "Error initializing
services.", e);
    }
}

private void doInitRemoting() {
    loadConnectionInfo();
    if (connectionInfo.isEnabled())
        initRemoteServer();
}

private void initRemoteConnectorServer() {
    remoteConnectorServer = new RemoteConnectorServer();
    remoteConnectorServer.setServiceRegistry(getServiceRegistry());
    remoteConnectorServer.setConnectionInfo(connectionInfo);
    remoteConnectorServer.bindToRegistry();
}
```

Und jetzt ohne Leerzeilen und mit hochgezogener schließender Klammer:

```
private void initServices(List<Service> services) {
    for (Service next : services)
        registerAndInitService(service); }
private void registerAndInitService(Service service) {
    if (service instanceof LifecycleSupport)
        ((LifecycleSupport) next).initialize();
    serviceRegistry.register(o); }
private void shutdownServices(List<Service> services) {
    for (Service next : services)
        shutdownService(service); }
private void shutdownService(Service service) {
    if (service instanceof LifecycleSupport)
        ((LifecycleSupport) service).shutdown(); }
protected void initPersistence() {
    PersistenceContextProvider provider = getPersistenceContextProvider()
    if (provider != null)
        provider.initialize(); }
protected void shutdownPersistence() {
    PersistenceContextProvider provider = getPersistenceContextProvider()
    if (provider != null)
        provider.shutdown(); }
private void initRemoting() {
    try {
        doInitRemoting();
    } catch (Exception e) {
        throw new EngineException("Initialization", "Error initializing
services.", e);
    }}
private void doInitRemoting() {
    loadConnectionInfo();
    if (connectionInfo.isEnabled())
        initRemoteServer(); }
private void initRemoteConnectorServer() {
    remoteConnectorServer = new RemoteConnectorServer();
    remoteConnectorServer.setServiceRegistry(getServiceRegistry());
    remoteConnectorServer.setConnectionInfo(connectionInfo);
    remoteConnectorServer.bindToRegistry(); }
```

Der Effekt sollte deutlich sein. Versuchen Sie trotzdem einmal, ihre Augen unfokussiert über beide Versionen gleiten zu lassen, um den Effekt noch zu verstärken.

Die umgekehrte Aussage gilt genauso: Was zusammengehört, sollte auch möglichst nicht voneinander getrennt werden, sei es durch Leerzeilen oder Kommentare. Relevant ist das insbesondere für Felder einer Klasse. Was wir im vorherigen Kapitel zu unnötigen Kommentaren gesagt haben, bekommt hier noch eine andere Begründung.

Die Kommentare zwischen den Feld-Definitionen reißen diese optisch auseinander. Das Ergebnis ist, dass sie nicht mehr als ein „Absatz“ aufgefasst werden, sondern als eigenständige Konzepte – was selten gewünscht sein dürfte.

7.3.5 Reihenfolgen

Auch bei der Reihenfolge orientieren wir uns an der Zeitungsmetapher:

Der allgemeine Teil sollte oben stehen, dann der speziellere Teil danach folgen. Auf Methoden bezogen bedeutet das:

Regel 7-2: Abstraktere Methoden stehen vor spezielleren Methoden. (Lesb)

Die Abstraktesten Methoden sind dabei natürlich die öffentlichen Methoden.

Regel 7-3: Abhängige Methoden sollten dicht zusammenstehen, dabei der Aufrufer (der abstraktere) über dem Aufgerufenen. (Lesb)

Schwierig wird das, wenn mehrere abstraktere Methoden sich spezielle Methoden teilen. Dann sollte die speziellere Methode unter beiden abstrakteren Methoden stehen.


Regel 7-4: Konzeptionell zusammengehörige Methoden sollten dicht beieinanderstehen. (Lesb)

Besitzt unsere Klasse die Methode `add()`, `insert()` und `addFirst()`, so sollten diese drei Methoden auch dicht beieinander definiert werden. Ein Sonderfall sind dabei die die Getter/Setter-Methoden, die alle zusammengehören und als Cluster an das Ende der Klasse geschrieben werden sollten

Eine letzte Regel lässt sich nicht auf die Zeitungsmetapher zurückführen, sondern eher auf ein Theaterstück:

Regel 7-5: Felder sollten vor Methoden definiert werden, Konstanten vor Feldern. (Lesb)

Fügen wir all diese Regeln zusammen, so haben wir ein Raster für den Aufbau unserer Klasse:

Klassenraster	
<ul style="list-style-type: none">▪ Konstanten▪ Felder▪ Schnittstellen-Methoden▪ Abstraktions-Methoden▪ Getter/Setter	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung
7 Seite 6	

7.3.6 Die Rubrik

Um unsere Metapher noch ein wenig weiter zu treiben: Nachrichtenartikel sind in der Regel nach Rubriken sortiert. Diese Aufgabe übernehmen bei uns die Pakete (oder Namensräume, oder Komponenten).

Wichtig dabei ist, dass natürlich auch die Rubrik einen vernünftigen Namen bekommt.


7.4 Weitere Formatierungsregeln

Nachdem wir oben die wichtigsten Regeln bereits definiert haben, wenden wir uns im Folgenden noch einigen weiteren Gesichtspunkten zu.

7.4.1 Breite und Höhe

Es gibt viele Meinungen und Thesen zur richtigen Länge einer Klasse und zur maximalen Breite ihrer Zeilen.

Statt uns in den Kampf einzumischen, bleiben wir lieber bei einigen Grundsätzen:

Ausmaße	
<ul style="list-style-type: none">▪ Klassen sollten in der Regel 100-200 Zeilen umfassen<ul style="list-style-type: none">▪ Das ist keine harte Grenze▪ Mehr als 500 Zeilen sollte eine Ausnahme sein▪ Die Breite sollte sich an den 80 Zeichen orientieren<ul style="list-style-type: none">▪ Das erlaubt einen vernünftigen Ausdruck▪ und bequem zwei Dateien nebeneinander auf einem Monitor zu betrachten	
<small>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH Clean Code - Professionelle Codeerstellung und Wartung 7 Seite 7</small>	

Da unsere Methoden kurz und unsere Klasse konzeptionell kurz sind, werden unsere Klassen i.d.R. nicht in eine Größenordnung kommen, bei der wir anfangen müssen, uns Gedanken zu machen. Wer trotzdem Zahlen möchte: das extremste Erlebnis des Autors war eine Klasse mit 12.000 Zeilen Code – extrem schlechter Code noch dazu: Dass das zu lang ist, sollte jedem klar sein.

In der Praxis hat sich ein Wert von 100-200 Zeilen als sinnvolle größere herausgestellt, wobei einige Klassen sicher noch ein ganzes Stück größer werden. 500 Zeilen sollten aber nur in Ausnahmefällen überschritten werden.

Was die Breite angeht: der alte Grundsatz mit der Bildschirmbreite hat auch heutzutage noch seine Berechtigung. Damit ist aber der alte Wert gemeint, also 80 Zeichen.

Ein Vorteil dieser Größenordnung ist, dass man den Code noch vernünftig ausdrucken kann, wenn es denn notwendig wird. Viel nützlicher ist aber die Tatsache, dass man damit zwei Dateien auf einem größeren Monitor bequem nebeneinander darstellen kann. Das erleichtert zum einen die Arbeit mit der Sourcecode-Verwaltung (Vergleiche zwischen Versionen), zum anderen aber auch das Nachverfolgen von Ausführungspfaden, die über mehr als eine Klasse gehen.

Die 80 Zeichen sind dabei keine harte Regel, sondern eine Richtschnur, die durchaus auch überschritten werden kann.

7.4.2 Einrückungen

Das Einrückungen Code deutlich lesbarer machen, sollte keine Überraschung mehr sein. Interessanterweise kann man viel Zeit damit verbringen, sich darüber zu streiten, wie groß diese sein sollten und ob Leerzeichen oder Tabulatoren verwendet werden sollten.

Weitere Gesichtspunkte	 integrata cegos	
<ul style="list-style-type: none">▪ Einrückungen sind wichtig für die Lesbarkeit<ul style="list-style-type: none">▪ Ob Tabulatoren oder Leerzeichen spielt dabei keine Rolle▪ Ausnahmen von Codeformatierungen sollten nur genutzt werden, wenn der eingesetzte Code-Formatter damit umgehen kann▪ Es ist müßig, sich über die Position von geschweiften Klammern zu streiten		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	7 Seite 8

Eigentlich sind diese beiden Fragen ziemlich müßig, wichtig ist nur, **dass** man einer Konvention folgt.

7.4.3 Ausnahmen

Jede Regel sollte Ausnahmen zulassen. Die Gefahr bei Ausnahmen ist aber, dass Code-Formatter, sollten sie denn eingesetzt werden, diese in der Regel nicht kennen und prompt wieder zurückformatieren.

Eine Ausnahme, die sich aus Sicht des Autors bewährt hat und auch von den meisten Formattern unterstützt wird ist folgende:

Ist die Handlung einer if/else Anweisung ein Exit-Punkt der Methode oder des Blockes, so kann diese in die gleiche Zeile geschrieben werden:

```
if (a > maxSize) return;

if (left == right) return 0;
else return -1;

if (number % j == 0) break;
```

7.5 Team Rules!

Wer ein wenig im Internet herumstöbert, wird ziemlich bald auf harte Fronten und schwere Grabenkriege zwischen Verfechtern unterschiedlicher Ansichten über Formatierungen stoßen.

Ganze Meetings sind schon an der Frage zerbrochen, ob nach der öffnenden Klammer einer Funktion ein Leerzeichen stehen soll, oder nicht.

Noch dramatischer sind die unterschiedlichen Ansichten über die Position der öffnenden geschweiften Klammern.²

Anstatt hier zu sehr ins Detail zu gehen und einen Style-Guide zu entwerfen, wollen wir direkt den einzig wahren Formatierungsstil definieren:

Regel 7-5: Der einzig wahre Formatierungsstil ist der, den das Team festgelegt hat. (Lesb, Vers, Wart)

Das bedeutet: Existieren keine Richtlinien, so ist es die Aufgabe des Teams, diese festzulegen und in Regeln für die IDE zu gießen. Das kann schon innerhalb von einer Viertelstunde passiert sein (also das Einigen – nicht das in die IDE einbringen).

² Stil-Name wie 1TBS für „The One True Brace Style“ (der einzig wahre Klammer Stil) deuten schon darauf hin, dass die Diskussion teilweise quasi-religiöse Formen angenommen hat – wobei die Bezeichnung 1TBS eher mit einem Augenzwinkern und als Kritik an der Diskussion an sich zu sehen ist.

Der geneigte Leser mag unter http://en.wikipedia.org/wiki/Indent_style einen Überblick bekommen

7.6 Zusammenfassung

In diesem Kapitel haben wir Grundsätze für die Formatierung von Sourcecode festgelegt. Wir haben eine Klasse mit einem Zeitungsartikel verglichen und daraus die wichtigsten Merkmale einer guten Struktur herausgearbeitet.

Zum Schluss haben wir uns mit der Frage nach der Wichtigkeit einiger gängiger Regeln beschäftigt.

Noch einmal:

Regel 7-6: Wichtig ist nicht, *welche* Formatierungsregeln im Einzelnen verwendet werden, sondern *dass* diese Regeln existieren und *von allen* genutzt werden. (Lesb, Vers, Wart)

8


Clean Code – Spezielle Themen

8.1	Einleitung.....	8-3
8.2	Exception Handling.....	8-4
8.2.1	Die Verwendung von Null	8-8
8.3	Nebenläufigkeit.....	8-9
8.3.1	Mythen und Missverständnisse.....	8-10
8.3.2	Die Herausforderung.....	8-12
8.3.3	Nebenläufige Prinzipien	8-14
8.3.4	Garbage Collector.....	8-17
8.3.5	Zusammenfassung	8-18
8.4	Optimierung.....	8-19
8.4.1	Einleitung	8-19
8.4.2	Was ist Performance?.....	8-19
8.4.3	Gefühlte Performance.....	8-20
8.4.4	Wann sollte optimiert werden?.....	8-21
8.4.5	Das Optimierungsdreieck.....	8-22
8.4.6	Zusammenfassung	8-23

8 Clean Code – Spezielle Themen


8.1 Einleitung

Dieses Kapitel beschäftigt sich mit verschiedenen speziellen Themen. Die Liste könnte deutlich länger sein. Teilweise sind die Themen auch abhängig von den verschiedenen Programmiersprachen. Hier eine Auswahl:

Einleitung	
<ul style="list-style-type: none">▪ Spezielle Themen im Clean Code Umfeld<ul style="list-style-type: none">▪ Liste könnte lang werden▪ Hier nur eine Auswahl▪ Exception Handling▪ Nebenläufigkeit (Concurrency)▪ Optimierung	
<div>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH</div> <div>Clean Code - Professionelle Codeerstellung und Wartung</div> <div>8 Seite 2</div>	

8.2 Exception Handling


Speziell in Java gibt es mittlerweile seit Jahren standardmäßig Exception Handling, welches man verwenden kann. Die erste Frage, die sich stellt, ist allerdings, ob Exception Handling oder Return Codes nutze. Es gibt fertige Systeme die Return Codes nutzen, wie beispielsweise IBM ACE Das Ganze ist bei Standardsystem ein gängiges Mittel und probates Mittel.

Exception Handling I		
<ul style="list-style-type: none">▪ Exceptions vs. Return Codes<ul style="list-style-type: none">▪ Einige fertige Systeme nutzen Return Codes z. B. IBM ACE, IBM MQ<ul style="list-style-type: none">▪ BIP2322E Problems connecting to a broker▪ return code= 2085 MQRC_UNKNOWN_OBJECT_NAME when trying to open a queue in the cluster▪ Bei fertigen Standard Systemen ein gutes Mittel▪ Bei eigenen Systemen nur möglich bei eigenem, vollständigen Konzept▪ Bei Standard Systemen ist der Nutzer der Return Codes der Administrator▪ Bei Individual Systemen ist der Nutzer der Exceptions Programmierer▪ Bei Individual Systemen sind die Exceptions erste Wahl<ul style="list-style-type: none">▪ Falls die Programmiersprache das unterstützt		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	8 Seite 3

Ein eigenes System zu schreiben, dass Reason Codes unterstützt bedarf eines vollständigen Konzepts. Wichtig dabei ist auch zu berücksichtigen, dass die Return Codes in der Regel für die Administratoren sind und die Exceptions für die Entwickler. Bei individuellen Systemen sind Exceptions, wenn die Programmiersprache das unterstützt, die erste Wahl.

Hier noch mal die wichtigsten Schlüsselwörter die im sechsten Exception Handy verwendet werden.

Exception Handling II



Exception Schlüsselwörter

- **try**
 - Definiert einen Block, innerhalb dessen Ausnahmen (Exceptions) auftreten können (geworfen werden können).
- **catch**
 - Definiert einen Block, der die Fehlerbehandlung für die durch den im catch-Befehl angegebenen Ausnahmetyp durchführt.
- **finally**
 - Definiert einen Block, der stets ausgeführt wird, egal ob ein Fehler auftrat oder nicht. Auch wenn
 - innerhalb des catch-Zweiges eine weitere Exception geworfen wird.
 - im catch-Zweig ein return steht.
 - kein catch-Zweig durchlaufen wird.
- **throw**
 - Erzeugt (wirft) eine Ausnahme. Hierfür muss dem Befehl throw ein Objekt übergeben werden, das eine Unterklasse von Throwable ist (dies sind die Klassen Exception und Error).
- **throws Exception**
 - Die Methode muss mit throws eine Liste aller Ausnahmen definieren, die geworfen werden können.

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung8 Seite 4

Beim Exception Handling gibt es generelle Regeln. Eine gute Idee ist es der Standard Ablauf zu definieren und dabei drauf zu achten, dass das wirklich den normalen Ablauf darstellt.

Exception Handling III





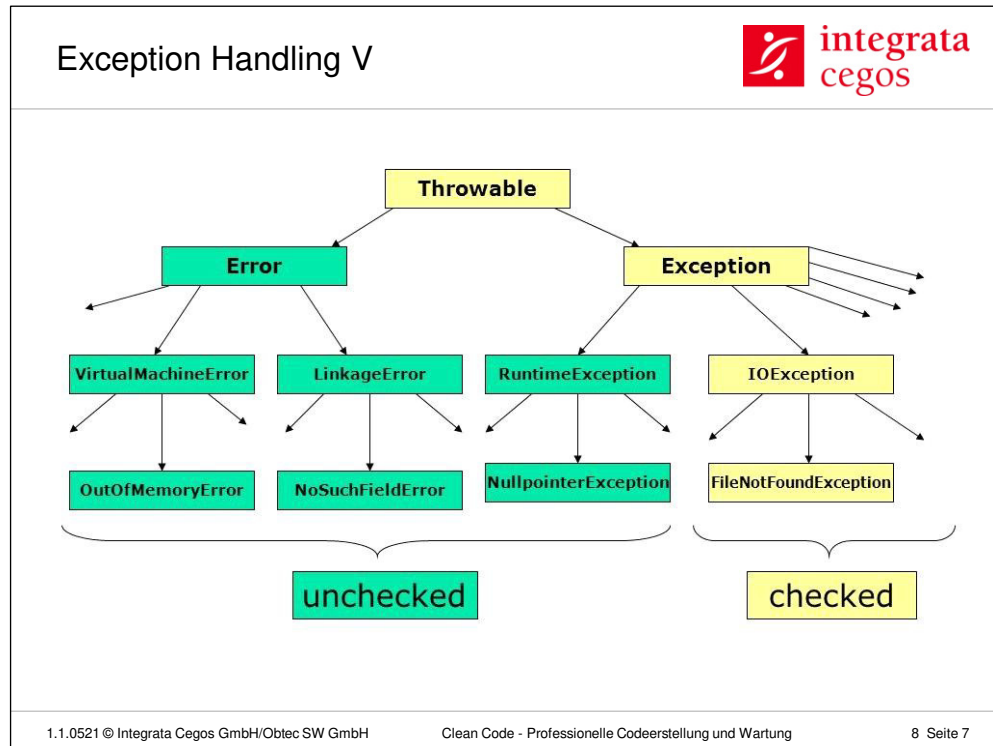
- Generelle Regeln
 - Festlegen des Standard Ablaufs
 - Write Your Try-Catch-Finally Statement First
 - Was ist der normale Ablauf?
 - Welche Ausnahmen gibt es?
 - Was muss auf alle Fälle passieren (zurücksetzen)?
 - z. B. JDBC Connection disconnect
- Use Unchecked Exceptions
- Provide Context with Exceptions

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbHClean Code - Professionelle Codeerstellung und Wartung8 Seite 5

Eine weitere gute Regel ist es sich erst mit dem Try Catch Finally Block zu beschäftigen. Es macht Sinn das Try Catch Finally Statement als erstes zu schreiben. Wichtig ist grundsätzlich immer den Finally festzulegen, was gerne vergessen wird. Dies führt z.B. bei Zugriffen über JDBC zu Performance Problemen.

Ferner muss man sich entscheiden, ob man so genannte unchecked Exceptions oder checked exceptions verwendet. Meist verwendet man heute an unchecked Exceptions. Sollte man eigene Exceptions definieren, sollte man ein Kontext mit den Exceptions mitgeben, die aus der Sicht des Aufrufers verständlich sind.


Exception Handling IV		
<p>Klassen zur Ausnahmebehandlung</p> 	<ul style="list-style-type: none">▪ Throwable (Superklasse von Error und Exception)▪ Error<ul style="list-style-type: none">▪ für fatale Fehler, die zur Beendigung des gesamten Programms führen.▪ Exception<ul style="list-style-type: none">▪ für behandelbare Fehler oder Ausnahmen.▪ RuntimeException (Subklasse von Exception)<ul style="list-style-type: none">▪ fasst die bei normaler Programmausführung evtl. auftretenden Ausnahmen zusammen. Sie kann jederzeit auftreten und kann, muss aber nicht abgefangen werden.▪ unchecked exception<ul style="list-style-type: none">▪ Ausnahmen der Klasse Error und RuntimeException müssen nicht im Methodenkopf deklariert werden.▪ checked exception<ul style="list-style-type: none">▪ Die anderen Ausnahmen, die in einer Methode auftreten können und dort nicht selbst abgefangen werden, müssen explizit im Kopf der Methode deklariert werden.	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH		Clean Code - Professionelle Codeerstellung und Wartung
		8 Seite 6



Eigene Exception Klassen sollte man eher für technische und nicht für logische Fehler nutzen. Der Vorteil der Nutzung bei technischen Fehlern ist, dass sie an der richtigen Stelle beispielsweise im Log File auftauchen. Logische Fehler sind an dieser Stelle eher fehl am Platz.

8.2.1 Die Verwendung von Null

Ein weiteres Thema was im Zusammenhang mit exceptions eine Rolle spielt, ist die Verwendung von Null.

Exception Handling VI	
<ul style="list-style-type: none">▪ Eigene Exception Klassen?<ul style="list-style-type: none">▪ Eher für technische Fehler als für logische Fehler nutzen▪ Bei der Definition von eigenen Exception Klassen die Sicht des Aufrufenden berücksichtigen▪ Don't Return Null<ul style="list-style-type: none">▪ Kann NullPointerException erzeugen und muss dann übers Exception Handling abgefangen werden▪ Muss also auf der aufrufenden Seite explizit berücksichtigt werden▪ Besser statt Null z.B. eine leere Liste zurück geben▪ Java:<ul style="list-style-type: none">▪ <pre>public List<Konten> getKonten() { if(.. Wenn da keinen Konten sind) return Collections.emptyList(); }</pre>▪ Don't Pass Null<ul style="list-style-type: none">▪ Lässt sich nur schwer verhindern und noch schwerer abzufangen▪ Ist Bad code, aber leider nicht zu verhindern	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 8 Seite 8

Man sollte weder Null übergeben, noch Null zurückgeben. Das Zurückgeben von Null hat den entscheidenden Nachteil, dass man auf der Aufrufen Seite Null abfangen muss, da es sonst eine Null Pointer Exception geben kann. Eine bessere Idee statt beispielsweise null zurück zu geben, wäre es eine leere Liste zurück zu liefern. Java unterstützt `Collections.emptyList()`.

Ferner sollte man auch keine Null übergeben. Das ist auf der Empfänger Seite schwer abzufangen. Das Übergeben von Null lässt sich nicht verhindern oder nur schwer verhindern, ist aber definitiv kein guter Code.

8.3 Nebenläufigkeit

Warum brauchen wir Nebenläufigkeit?

Nebenläufigkeit bedeutet, dass mehrere Routinen quasi-gleichzeitig, weitestgehend voneinander getrennt, ablaufen. Welche Vorteile hat das für uns?

- **Verbesserung von Struktur und Design**

Hängen die einzelnen Aufgaben nicht wirklich zusammen, so können sie vollständig voneinander getrennt werden. Jede Aufgabe verhält sich damit, als würde sie vollständig für sich alleine ausgeführt werden.

Ein Spezialfall dieser Begründung sind Server-Anwendungen, die eine Anzahl an Nutzern gleichzeitig bedienen, ohne dass sich die einzelnen Nutzer in die Quere kommen.

- **Nutzen von Wartezeiten**

Ein Programm, das in irgendeiner Form I/O betreibt, verbringt einen Großteil seiner Laufzeit in einem wartenden Zustand. In einer Single-Thread-Applikation ist diese Zeit verloren. In einer Nebenläufigen Anwendung kann stattdessen während der Wartezeit ein anderer Thread weiterarbeiten. Das kann einen immensen Performancegewinn bedeuten.

- **Ausnutzung der Hardware**

Bis vor wenigen Jahren bedeutete Weiterentwicklung der Hardware hauptsächlich eine vertikale Skalierung, dass heißt die Transistoren wurden immer kleiner, ihre Dichte immer höher, und damit Prozessoren immer schneller.¹

Seit einigen Jahren ist allerdings eine Richtungsänderung zu erkennen. Statt immer mehr Transistoren in einem Prozessor unterzubringen, bestehen die Prozessoren aus einzelnen Kernen (vier Kerne sind für neue Standard-PCs schon nicht ungewöhnlich), die in der Lage sind, Code echt nebenläufig auszuführen.

Das hat einen fundamentalen Einfluss auf unsere Programme. Sind diese nämlich nicht auf Nebenläufigkeit ausgelegt, so nutzen sie auch nur einen einzigen Prozessorkern!

- **Nutzerfeedback**

Wenn ein Nutzer in einer Oberfläche eine Aktion initiiert, dann erwartet er auch zeitnah eine Reaktion. Dauert die Aktion aber länger, so sollte der Nutzer zumindest darüber informiert werden, dass das Sys-

¹ Moore's Law, der Versuch, die Weiterentwicklung der Leistungsfähigkeit von Prozessoren vorauszusagen, besagt, dass sich die Anzahl der Transistoren alle zwei Jahre ungefähr verdoppelt. Seit seiner Formulierung (1965!) hat sich die Vorhersage bewahrheitet.

tem beschäftigt ist (durch die Sanduhr als Mauszeiger oder einen Fortschrittsbalken). Diese Benachrichtigung wird in einem separaten Thread stattfinden.

8.3.1 Mythen und Missverständnisse

Es existieren einige Mythen und Missverständnisse zur Nebenläufigkeit, die immer wieder auftauchen:

Nebenläufigkeit - Mythen und Missverständnisse	 integrata cegos	
<ul style="list-style-type: none">▪ Nebenläufigkeit verbessert grundsätzlich die Performance▪ Das Design ändert sich nicht durch die Nebenläufigkeit▪ Durch den Einsatz eines Frameworks erspart man sich das Auseinandersetzen mit Nebenläufigen Prinzipien		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	8 Seite 9

- *Nebenläufigkeit verbessert grundsätzlich die Performance*

Natürlich kann Nebenläufigkeit die Performance verbessern, damit beschäftigen sich ja auch 2 von 4 Begründungen in der Einleitung. Aber zum Einen gibt es ja noch die anderen beiden Begründungen, zum Anderen muss das Problem auch aufteilbar sein. Selbst dann bedeutet Nebenläufigkeit immer auch zusätzlichen Verwaltungsaufwand, so dass im schlimmsten Fall eine Verlangsamung eintritt.

- *Das Design ändert sich nicht durch die Nebenläufigkeit*


Tatsächlich ist das Design oftmals sehr unterschiedlich im Vergleich zu einer Single-Threaded Applikation – aber nicht notwendigerweise komplizierter.

- *Durch den Einsatz eines Frameworks erspart man sich das Auseinandersetzen mit Nebenläufigen Prinzipien*

Leider ein Mythos, der sich hartnäckig hält. Was uns das Framework abnimmt, ist das Arbeiten in der Schlamzone, das Erstellen von Threads etc. Trotzdem sind wir dafür verantwortlich, dass unsere Klassen auch miteinander zurechtkommen.


Wahrheiten

Im Gegensatz dazu folgen nun ein paar Wahrheiten:

Wahrheiten	
<ul style="list-style-type: none">▪ Nebenläufigkeit bedeutet immer zusätzlichen Aufwand<ul style="list-style-type: none">▪ Sowohl in der Leistung, durch zusätzliche Verwaltung, als auch in der Programmierung▪ Korrekte Nebenläufigkeit ist komplex<ul style="list-style-type: none">▪ Auch, wenn es das eigentliche Problem nicht ist.▪ Nebenläufige Bugs sind nicht oder nur kaum reproduzierbar<ul style="list-style-type: none">▪ Mit dem Ergebnis, dass sie häufig als Kuriositäten („Sonnenflecken“) abgetan und ignoriert werden.▪ Nebenläufigkeit erfordert häufig eine fundamentale Änderung der Design Strategie.	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 8 Seite 10

8.3.2 Die Herausforderung

Betrachten wir folgendes, kleines Programm:

Das Problem		
<pre>public class Raiser { private int lastValue; public int getNextValue() { return ++lastValue; } }</pre>		
<ul style="list-style-type: none">▪ Bei gleichzeitigem Aufruf von zwei Threads gibt es drei mögliche Endzustände:<ul style="list-style-type: none">▪ Thread 1 bekommt den Wert 1, Thread 2 den Wert 2, lastValue hat den Wert 2▪ Thread 1 bekommt den Wert 2, Thread 2 den Wert 1, lastValue hat den Wert 2▪ Thread 1 bekommt den Wert 1, Thread 2 den Wert 1, lastValue hat den Wert 1		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	8 Seite 11

Die beiden ersten Ergebnisse sind natürlich die, die wir erwartet haben, aber das dritte Ergebnis ist ungewöhnlich – bzw. sogar gefährlich, wenn in unserem Programm jeder Thread eine eindeutige ID haben muss!

Wo liegt hier das Problem? Natürlich im Ausdruck `++lastValue`. Ausgeschrieben (und mit einer temporären Variable besser lesbar gemacht) steht dort:

Was kann hier nun schiefgehen? Betrachten wir die beiden Threads mit einander verschränkt. Beide besitzen eine voneinander unabhängige Variable `temp`: Das Problem ist, dass der Threadwechsel zwischen dem Auslesen und dem Zurückschreiben von `lastValue` passieren kann.

Das Problem



- ++lastValue ausgeschrieben:

```
public int getNextValue() {  
    int temp = lastValue;  
    lastValue = temp + 1;  
    return lastValue;  
}
```
- Mögliches Szenario für das Scheitern:
 - lastValue: 0
 - int temp_1 = lastValue; -> temp_1: 0
 - Threadwechsel
 - int temp_2 = lastValue; -> temp_2: 0
 - lastValue = temp_2 + 1 -> lastValue: 0 + 1 = 1
 - return lastValue -> Rückgabe: 1 (für Thread 2)
 - Threadwechsel
 - lastValue = temp_1 + 1 -> lastValue: 0 + 1 = 1
 - return lastValue -> Rückgabe: 1 (für Thread 1)

1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH

Clean Code - Professionelle Codeerstellung und Wartung

8 Seite 12

Dieses Problem nennen wir *Read-Write-Problem*. Ein Wert wird ausgelesen und abhängig von dem gelesenen Wert wird ein anderer (oder derselbe) gesetzt. Es kann nur dadurch gelöst werden, dass verhindert wird, dass während des Read-Write Vorganges ein anderer Thread denselben Code auf derselben Instanz ausführt – mit anderen Worten, der Read-Write-Zugriff muss *atomar* erfolgen.

8.3.3 Nebenläufige Prinzipien

Wir beginnen mit einigen Prinzipien, die für nebenläufige Programmierung gelten:

Nebenläufige Grundsätze	 integrata cegos
<ul style="list-style-type: none">▪ Jede Klasse muss dokumentieren, ob sie Thread-sicher ist<ul style="list-style-type: none">▪ Read Only Klassen sind inherent Thread-sicher▪ Thread-sichere Klassen<ul style="list-style-type: none">▪ Klassen, die dafür ausgelegt sind, in mehreren Threads gleichzeitig genutzt zu werden (und sich selbst um Synchronisierung kümmern)▪ Nicht Thread-sicher<ul style="list-style-type: none">▪ Eine Klasse, die nicht Thread-sicher ist, muss das deutlich dokumentieren, ggf. mit dem Hinweis, was nötig ist, um die Klasse Thread-sicher zu machen▪ Partiiell Thread-sicher<ul style="list-style-type: none">▪ Die Klasse ist mit (genau dokumentierten) Einschränkungen Thread-sicher	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung
8 Seite 13	

Nebenläufig oder nicht


Wenn wir eine Klasse schreiben, sollten wir uns von vorne herein Gedanken darüber machen, wie sie sich in einem nebenläufigen Szenario verhält. Diese Gedanken müssen dokumentiert werden!

Einige Grundsätze:

Unveränderliche Objekte sind Thread-Sicher. Ein Objekt ist dann unveränderlich, wenn es keine Möglichkeit mehr gibt, nach seiner Erstellung seinen Zustand zu verändern (und idealerweise sind alle Attribute als *final* deklariert, wenn die Sprache ein derartiges Konstrukt kennt).

Ist unsere Klasse veränderbar, so müssen wir uns entscheiden: Wie wahrscheinlich ist es, dass eine Instanz dieses Objektes zwischen Threads ausgetauscht wird? Ist das eine realistische Situation, so müssen wir dafür sorgen, dass das Objekt Thread-sicher wird. Kommt die Situation nicht in Frage, so können wir die Klasse Thread-unsicher lassen, müssen das aber deutlich dokumentieren.

Regel 8-1: In jeder Klasse muss beschrieben sein, ob diese Thread-sicher ist oder nicht. Unveränderliche Objekte sind immer Thread-sicher. (Lesb, Vers)

Atomare Zugriffe	
<ul style="list-style-type: none">▪ Bestimmte Zugriffe müssen atomar erfolgen<ul style="list-style-type: none">▪ Read-Write Zugriffe<ul style="list-style-type: none">▪ Diese treten auf, wenn eine Schreiboperation auf eine Variable von einer vorangegangenen Leseoperation abhängig ist▪ Dependent Writes<ul style="list-style-type: none">▪ Mehr als ein Feld wird geschrieben▪ Die Felder stehen fachlich in Abhängigkeit zu einander (z.B. über eine Invariante)▪ Zugriffe auf „lange Variablen“ (long, double)<ul style="list-style-type: none">▪ Je nach Betriebssystem und Architektur können diese Zugriffe auch als zwei getrennte Zugriffe erfolgen	
<small>1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH</small>	<small>Clean Code - Professionelle Codeerstellung und Wartung 8 Seite 15</small>

Regel 8-2: *Read-Write-Zugriffe* und *Dependent-Writes* müssen atomar ausgeführt werden.

Regel 8-3: Felder die einmal geschützt werden, müssen *immer* geschützt werden.

Das heißt, auch die Lesezugriffe müssen geschützt werden!

Das Single-Responsibility-Principle

Dieses Prinzip haben wir ja schon ausführlich behandelt. Es besagt, dass es nur einen Grund geben darf, um eine Klasse oder Methode zu ändern.

Nebenläufigkeit ist eindeutig einer dieser Gründe. Das bedeutet für uns:

Regel 8-4: Code, der sich mit Nebenläufigkeit beschäftigt, sollte von anderem Code getrennt gehalten werden. (Lesb, Vers)

Insbesondere ist unser Nebenläufigkeitscode ja technischer Code, und den wollen wir ja sowieso von unserem fachlichen Code trennen.

Begrenzte Schreibzugriffe

Regel 8-5: Müssen wir den Zugriff auf ein oder mehr Felder schützen, so sollte der Zugriff auf diese Felder an so wenig Stellen wie möglich erfolgen.

Je häufiger wir auf die Felder zugreifen, desto größer ist die Chance, dass wir die Synchronisation vergessen.

Daten-Kopien und unabhängige Threads

Oftmals ist der einfachste Weg, mit geteilten Daten umzugehen, sie gar nicht erst zu teilen. Anstatt ein gemeinsames Objekt zu verteilen, bekommt jeder Thread seine eigene Kopie, die dann ggf. zum Ende der Threads wieder vereint werden müssen.

Weitere Grundsätze



- Nebenläufigkeit ist ein Änderungsgrund im Sinne des Single-Responsibility-Principles
- Daten-Kopien
 - Ein einfacher Weg mit geteilten Daten umzugehen ist es, einfach Kopien der Daten herzustellen, und diese zu verteilen
 - Ggf. müssen die Kopien am Ende der Verteilung wieder zusammengefügt werden, was viel Aufwand bedeuten kann.
- Unabhängige Threads
 - Threads sollten so unabhängig wie möglich sein
 - Eine Möglichkeit ist auf Zustände weitestgehend zu verzichten und den „Zustand“ als Parameter für alle Methoden zu übergeben

Das ist auch das Konzept funktionaler Programmierung. Datenstrukturen werden nicht verändert. Wird einer Liste ein Element hinzugefügt, wird stattdessen eine neue Liste mit den Elementen der alten plus dem neuen Element erzeugt.

Threads sollten so unabhängig wie möglich sein. Eine Möglichkeit das zu erreichen, ist den Objekten, die mehrfach genutzt werden, überhaupt keinen Zustand zu geben. Stattdessen werden alle Informationen als Parameter übergeben (was natürlich eine genaue Umkehr unserer Definition von guten Methoden ist).

Damit sind alle Variablen nur lokal, und somit auch nur im aktuellen Thread überhaupt nutzbar.

Diese Form ist für Server-Anwendungen (z.B. Webserver) sehr gut geeignet.

8.3.4 Garbage Collector

Ein weiteres Thema ist die Verwendung eines expliziten Aufrufs des Garbage Collectors.

Garbage Collector



- kettet unbenutzten Objekt-Speicher wieder in die Speicherkette.
- zerstört Objekte bzw. räumt den heap-Speicher auf.
- kann nicht direkt aufgerufen werden sondern läuft als unabhängiger Thread.
- kann angestoßen werden (triggern) durch `System.gc()`.
- Bevor ein Object endgültig zerstört wird, wird die in Object definierte `finalize()`-Methode (seit JDK 9 deprecated) aufgerufen.
 - `protected void finalize() throws Throwable`

8.3.5 Zusammenfassung

Threads sind kompliziert. Und dieses Kapitel ist keinesfalls ein Ersatz für eine umfassende Beschäftigung mit dem Thema. Aber es ist ein erster Ansatz, der uns auf die eigentliche Arbeit mit Threads zumindest vorbereitet.

Zusammenfassung



- Nebenläufigkeit sollte nicht schlechten Code ausgleichen
- Nebenläufigkeit sollte sauber geplant werden
- Nebenläufigkeit eher in Ausnahmefällen verwenden
- Für den expliziten Garbage Collector Aufruf gilt selbiges

8.4 Optimierung


8.4.1 Einleitung

In diesem Kapitel wollen wir einen kurzen Überblick über die Grundsätze des Optimierens gewinnen. Wir beschäftigen uns zunächst mit dem Begriff der Performance um anschließend einen allgemeinen Prozess für die Optimierung zu definieren.

8.4.2 Was ist Performance?

Die Performance eines Softwaresystems oder einer Softwarekomponente wird von Connie U. Smith und Loyd G. Williams² wie folgt definiert.

„Performance is an Indicator of how well a software system or component meets its requirements for timeliness.“

Optimierung	
<p><i>„Performance is an Indicator of how well a software system or component meets its requirements for timeliness.“</i></p> <ul style="list-style-type: none">▪ <i>Kenngrößen der Performance sind</i><ul style="list-style-type: none">▪ <i>Ressourcen Nutzung (CPU, Speicher, IO)</i>▪ <i>Antwortzeiten</i>	
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung 8 Seite 20

Als Messgrößen für diese „Rechtzeitigkeit“ werden oft die Antwortzeit bzw. der Durchsatz eines Softwaresystems benutzt. Die Antwortzeit ist die Zeit, die aus Sicht des Benutzers der Software vergeht, bis eine bestimmte Anfrage oder Aufgabe bearbeitet wurde. Der Durchsatz gibt an, wie viele Aufgaben (Transaktionen, Datensätze, Daten etc.) von der Software in einem bestimmten Zeitraum verarbeitet werden können.

² Connie U. Smith, Loyd G. Williams – Performance Solutions – A practical Guide to Creating Responsive, Scalable Software

Eng verbunden mit der Antwortzeit bzw. dem Durchsatz eines Softwaresystems ist seine Skalierbarkeit, d.h. wie verhalten sich diese zeitlichen Werte, wenn die Last, die das System zu bewältigen hat, ansteigt. Prinzipiell ist kein Softwaresystem beliebig skalierbar. Ab einem bestimmten Punkt wird eine minimale Erhöhung der Last eine exponentielle Auswirkung auf Antwortzeit und Durchsatz haben. Entscheidend dabei ist, ob dieser Punkt bereits durch Lasten erreicht wird, die laut Anforderungen noch im Bereich der zu erwartenden Lasten für das reale System liegen oder außerhalb. Im ersten Fall hat das System seine Performanceanforderungen nicht erfüllt und damit ein Performanceproblem.

Performanceprobleme resultieren meist daraus, dass eine oder mehrere benötigte Ressourcen, wie z.B. Prozessoren, Speicher, Netzwerkbandbreite usw., nicht bzw. nicht in ausreichender Anzahl oder Menge zur Verfügung stehen. Eine Möglichkeit diesen Performanceproblemen zu begegnen ist es, der Anwendung zusätzliche oder bessere Ressourcen zur Verfügung zu stellen. Die andere Möglichkeit besteht darin, die Nutzung der vorhandenen Ressourcen zu optimieren, um die Performance eines Softwaresystems zu verbessern. Dieses Vorgehen bezeichnet man als Performancetuning.

8.4.3 Gefühlte Performance

Neben der über Antwortzeiten bzw. den Durchsatz messbaren Performance eines Softwaresystems gibt es im Bereich der Software, die durch Benutzer bedient wird, noch den Begriff der gefühlten Performance. Diese subjektive Wahrnehmung des Benutzers kann dazu führen, dass eine Anwendung, welche eine höhere Antwortzeit als eine vergleichbare andere Anwendung hat, trotzdem als die performantere Variante wahrgenommen wird.

Damit ein Benutzer diese subjektive Empfindung einer performanten Anwendung bekommt, ist es notwendig, die Oberflächen entsprechend reaktiv zu gestalten. Dazu gehört die Darstellung des Arbeitsfortschritts durch Fortschrittsbalken oder ähnliches bzw. die Präsentation von Teilergebnissen, aber auch die Möglichkeit noch nicht beendete Operationen jederzeit abbrechen zu können. Operationen mit langer Laufzeit werden am besten in den Hintergrund verlegt, damit der Benutzer in dieser Zeit bereits andere Aufgaben wahrnehmen kann. Dem Benutzer muss ein Gefühl der Kontrolle über die Anwendung vermittelt werden.

8.4.4 Wann sollte optimiert werden?

Die größte Gefahr bei der Optimierung geht von einem vorschnellen Handeln aus. Viel Zeit geht in der Regel darauf verloren, dass Entwickler vermeintliche Flaschenhälse optimieren, die in der Praxis überhaupt keine Auswirkungen auf die Performance haben.

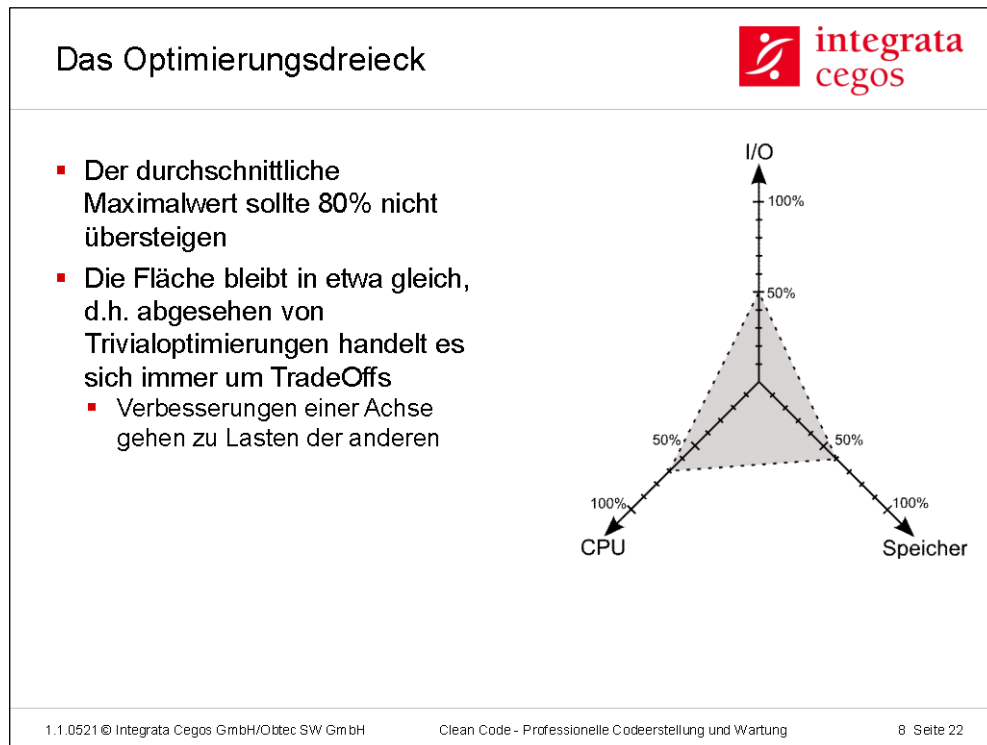
Optimiert wird dann, wenn es konkrete Forderungen dazu gibt. Entweder als Teil des Pflichtenheftes oder als Fehlerticket.

Bevor man als Entwickler also selbst den Quellcode „optimiert“, muss man sicherstellen, dass überhaupt ein Performanceproblem existiert und dass die gewählte Codeoptimierung auch passend für das Problem ist. Die Optimierung ist natürlich entsprechend zu dokumentieren und zu testen, d.h. werden die Performancekriterien auch nach der Optimierung erfüllt oder haben sich neue Problemfelder ergeben.

Wann sollte optimiert werden?	 integrata cegos	
<p>First Rule of Program Optimization – Don't do it. Second Rule of Program Optimization (for experts only!) – Don't do it yet.</p> <ul style="list-style-type: none">▪ Es soll erst optimiert werden, wenn es einen klaren Grund dafür gibt<ul style="list-style-type: none">▪ Nicht funktionale Anforderungen▪ „gefühlte Probleme“▪ Ressourcen-Engpässe		
1.1.0521 © Integrata Cegos GmbH/Obtec SW GmbH	Clean Code - Professionelle Codeerstellung und Wartung	8 Seite 21

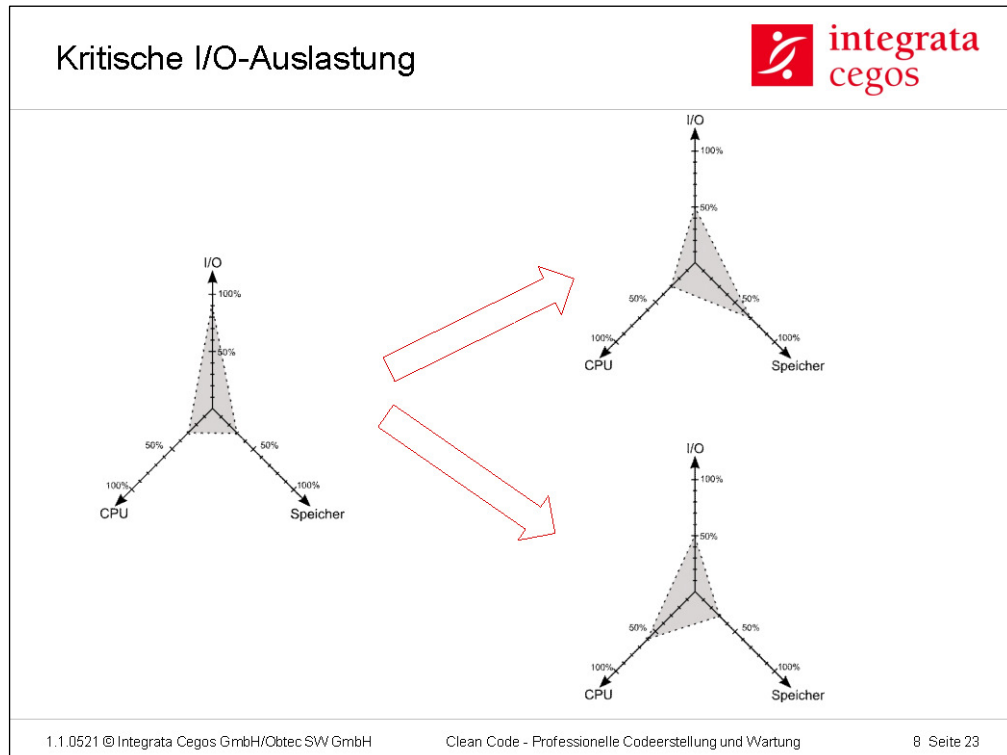
8.4.5 Das Optimierungsdreieck

Jede Anwendung stützt sich im Grundprinzip auf die drei Kenngrößen I/O-, CPU- und Speicherauslastung. Diese bilden die Achsen eines Dreiecks, dessen Fläche den Ressourcenverbrauch der Anwendung repräsentiert.



Der durchschnittliche Maximalwert der drei Auslastungen sollte dabei 80% nicht übersteigen. So bleibt noch Spielraum für eventuell auftretende Spitzen. Die Fläche des Optimierungsdreiecks bleibt bei fast allen Optimierungen – sieht man einmal von Trivialoptimierungen ab – nahezu konstant. Das bedeutet, dass fast alle Optimierungen sogenannte Trade-Off Optimierungen sind. Die Verbesserung von einem Wert führt zu einer entsprechenden Verschlechterung bei einem oder beiden der anderen Werte.

Um die I/O Auslastung zu senken müssen entweder die Anzahl der I/O Aufrufe oder die zu übertragenden Daten reduziert werden. Eine gebräuchliche Methode für die Reduzierung von I/O Aufrufen ist das Caching. Dabei werden Daten für den mehrmaligen Gebrauch zwischengespeichert, anstatt sie immer wieder neu über I/O Aufrufe abzufragen. Caching reduziert die I/O-Auslastung hat allerdings einen höheren Speicherbedarf, da die Daten im Regelfall im Speicher gehalten werden. Das Dreieck verschiebt sich also wie folgt.



Um die zu übertragenden Daten zu reduzieren, wird oft eine Kompression der Daten durchgeführt. Die Kompressionsalgorithmen bedingen allerdings eine höhere CPU-Auslastung. Befinden sich alle drei Auslastungen nahe am bzw. bereits im kritischen Bereich, wird eine normale Optimierung nicht mehr helfen. In diesem Fall müssen der Anwendung zusätzliche Ressourcen zur Verfügung gestellt werden und die Achsen somit wieder „verlängert“ werden.

Eine Erweiterung des Optimierungsdreiecks stellt die Optimierungspyramide dar. Dabei wird als vierte Größe die Wartbarkeit einer Anwendung betrachtet. Durch die Pyramide wird der Einfluss von Optimierungen auf die Lesbarkeit des Quellcodes und damit auf die Kosten für die zukünftige Pflege und Erweiterung der Anwendung visualisiert. Gerade Triviale Optimierungen, die im normalen Optimierungsdreieck die Fläche verkleinern, zeichnen sich oft durch sehr unübersichtlichen und schwer zu verstehenden Quellcode aus, der im Endeffekt zu erhöhten Pflegeaufwand und damit verbundenen Kosten führt.

8.4.6 Zusammenfassung

Optimierung ist ein weites Feld. Wir haben uns hier bewusst nicht auf konkrete Optimierungstechniken bezogen sondern uns stattdessen mit dem allgemeinen Optimierungsprozess beschäftigt.

Die einzige Regel für dieses Kapitel steht daher am Schluss:

Regel 8-6: Bevor optimiert wird muss es ein quantifizierbares Ziel und einen dazu passenden, identifizierten Engpass geben.

9

Metriken

9.1	Einleitung.....	9-3
9.1.1	Code Entropy.....	9-3
9.1.2	Die Zeitachse.....	9-3
9.2	Basis Metriken.....	9-4
9.2.1	Cyclomatic Complexity (CC).....	9-4
9.2.2	Lines of Code (LOC).....	9-5
9.2.3	Non Commenting Source Statements (NCSS).....	9-6
9.3	Objektorientierte Metriken.....	9-7
9.3.1	Weighted Methods per Class (WMC).....	9-7
9.3.2	Depth of Inheritance Tree (DIT).....	9-7
9.3.3	Number of Children (NOC).....	9-7
9.3.4	Coupling between Object Classes (CBO).....	9-7
9.3.5	Response for a Class (RFC).....	9-8
9.3.6	Lack of Cohesion in Methods (LCOM).....	9-8
9.3.7	Bewertung.....	9-9
9.4	Statische Analyse Tools (Bug Finder).....	9-10
9.5	Laufzeit Metriken.....	9-11
9.5.1	Testabdeckung.....	9-11
9.5.2	Builddauer.....	9-11
9.6	Zusammenfassung.....	9-12

9 Metriken

9.1 Einleitung

In diesem Kapitel beschäftigen wir uns mit einigen Metriken, um die Qualität unsere Software zu messen. Metriken sind gleichzeitig ein Fluch und ein Segen.

Korrekt angewendet können uns Metriken frühzeitig vor auftretenden Problemen warnen und uns die Möglichkeit geben, zu reagieren, bevor der Code zu weit degeneriert.

Uninterpretiertes Auswerten der Metriken führt dagegen zum „Coding against Metrics“-Phänomen, bei dem nicht mehr das Ziel des saubersten Codes, sondern der saubersten Metriken an oberster Stelle steht.

9.1.1 Code Entropy

Es ist ein Fakt, dass Code mit der Zeit degeneriert. So klar die ursprüngliche Architektur und das Design auch sein mögen, kleinere Änderungen an Anforderungen und nachträgliche Korrekturen führen dazu, dass der Code aus Sicht der intrinsischen Qualitätsmerkmale immer schlechter wird.

Dieses Entfernen von der Ideallinie bezeichnen wir als Code Entropy.

9.1.2 Die Zeitachse

Metriken stehen nie für sich alleine. Tatsächlich bekommen Sie erst dann eine wirkliche Aussagekraft, wenn man sie über einen Zeitraum betrachtet. Das bedeutet, die Ergebnisse müssen in ein Diagramm eingetragen werden, und zwar idealerweise in regelmäßigen Abständen (jede Nacht) und automatisiert.

Ergibt sich in so einem Diagramm eine Kurve, die immer weiter steigt, so können wir daran erkennen, dass unsere Qualität langsam degeneriert - dann wird es Zeit, durch gezielte, Qualitätsfördernde Maßnahmen gegen das Problem vorzugehen, z.B. durch eine Refactoring-Runde.

9.2 Basis Metriken

Zunächst wenden wir uns einigen Metriken zu, die sich nicht gezielt auf Objektorientierte Programme beziehen, sondern auch in der prozeduralen Programmierung Sinn machen (können).

9.2.1 Cyclomatic Complexity (CC)

Die Cyclomatic Complexity (zyklomatische Zahl nach McCabe) beschreibt, wie komplex eine Routine ist. Grob gesagt zählt sie, wie viele unabhängige Pfade es durch die Routine gibt.

Man bestimmt die CC für eine Methode in der Praxis, indem man von 1 ausgeht und dann für jeden Entscheidungspunkt die Zahl um eins erhöht.

Entscheidungspunkte sind dabei Verzweigungen (if), Schleifen (for und while) und Pfade in Entscheidungstabellen (switch). In einer switch Anweisung erhöht also jeder case-Pfad den CC um eins, was dazu führt, dass Methoden, die dieses Konstrukt verwenden, in der Regel recht hohe CCs besitzen.

Beispiel:

```
public void countCC() {  
    if ( c1() )  
        f1();  
    else  
        f2();  
  
    if ( c2() )  
        f3();  
    else  
        f4();  
}
```

Diese Methode besitzt einen CC von 3 (1 + jeweils 1 für die beiden if-Abfragen).

Diese Zählweise gilt allerdings nur für Methoden mit genau einem Eingangs- und einem Ausgangspunkt. Existiert mehr als ein Ausgangspunkt, so wird folgende Formel angewandt:

$$CC = \pi - s + 2$$

Wobei π die Anzahl der Entscheidungspunkte und s die Anzahl der Ausgangspunkte ist.

```
public int countCC2(int x, int y) {  
    if (x == y) return 0;  
    int z = 0;  
    while (x > y) {  
        z++;  
        x -= y;  
    }  
    if (x == 0) throw new IllegalStateException();  
    return z;  
}
```

Der Code hat drei Entscheidungs- und zwei Exit-Punkte, dementsprechend ist der CC dieser Methode $3 - 2 + 2 = 3$.

Der Erweiterte CC (CC') geht noch einen Schritt weiter in dem er auch das Vorhandensein der Short-Circuit-Operatoren `||` und `&&` als Entscheidungspunkte wertet.

Der CC ist eine relative nützliche Metrik. Zunächst gibt er uns eine Aussage darüber, wie viele Testfälle mindestens nötig sind, um die Methode abdecken zu können. Weiterhin gibt er uns einen guten Überblick über den Zustand unseres Codes an sich, allerdings nur, wenn man ihn auch sinnvoll verwendet.

Eine sinnvolle Anwendung, die Praxis-relevante Ergebnisse liefert, ist den gemittelten CC über die 10 längsten Methoden im Code zu liefern – bzw. einfach die 10 höchsten CCs zu mitteln.

Gängige Konventionen sind, dass der CC in der Regel unter 10, in Ausnahmefällen auch bis 20 gehen darf.

Halten wir uns an den in dieser Unterlage propagierten Code-Stil, wird der CC allerdings deutlich niedriger, in höchstens 7 Codezeilen wird es ziemlich schwierig einen CC über 5 unterzubringen.

Trotzdem sollte der CC natürlich im Auge behalten werden. Sinnvolle Maßnahmen, die man ergreifen kann, wenn der CC zu hoch wird, sind natürlich das Zerlegen der Methode in Einzelmethoden.

9.2.2 Lines of Code (LOC)

Eine gefährliche Metrik ist Lines of Code. Grundsätzlich bedeutet es, alle Zeilen zu zählen und aufzuaddieren. Das Ergebnis ist ein Wert für eine Klasse, ein Paket oder ein komplettes Projekt. Je nach tatsächlicher Zählweise werden Leerzeilen dabei entweder voll, teilweise oder gar nicht gezählt.

Die Aussagekraft von LOC ist leider sehr gering, da insbesondere auch Kommentare mitgezählt werden. Es dient eigentlich nur dazu, im Auge

zu behalten, wie schnell oder wie stark der Code wächst – und dient als interner Meilenstein („wir haben die 100.000 Zeilen geknackt!“)

Nützlich kann LOC auch sein, sehr grob Projekte mit einander zu vergleichen (ein LOC 10.000 ist sicher deutlich einfacher als ein LOC 100.000), wobei die Werte sich schon um Größenordnungen unterscheiden müssen, damit der Vergleich etwas bringt.

Der einzige nennenswerte Vorteil, den LOC bringt, ist, dass er extrem einfach zu bestimmen und zu verstehen ist.

Der Nachteil und die große Gefahr bestehen dabei darin, dass das Management ggf. auf die Idee kommen könnte, diese Metrik zur Bewertung von Leistungen zu nutzen.

9.2.3 Non Commenting Source Statements (NCSS)

Eine etwas bessere Alternative zu LOC sind die Non Commenting Source Statements, eine Metrik die nur die tatsächlichen Anweisungen zählt, also keine Leerzeilen oder Kommentare. Dabei wird der Zähler für jedes Statement einfach um eins erhöht. Unter Java wären das beispielsweise folgende Code-Elemente:

	Beispiel	Kommentar
Paket Deklaration	<code>package java.lang;</code>	
Import declaration	<code>import java.awt.*;</code>	
Class declaration	<code>-public class Foo { -public class Foo extends Bla {</code>	
Interface declaration	<code>public interface Able {</code>	
Field declaration	<code>-int a; -int a, b, c = 5, d = 6;</code>	
Method declaration	<code>-public void cry(); -public void gib() throws DeadException {</code>	
Constructor declaration	<code>public Foo() {</code>	
Constructor invocation	<code>-this(); -super();</code>	
Statement	<code>-i = 0; -if (ok) -if (exit) { -if (3 == 4); -if (4 == 4) { ; } -} else {</code>	expression, if, else, while, do, for, switch, break, continue, return, throw, synchronized, catch, finally
Label	<code>fine :</code>	normal, case, default

NCSS liefert als Metrik schon relative brauchbare Ergebnisse. Hat eine Methode einen doppelt so hohen NCSS wie eine andere, so ist sie in der Regel auch gefühlt doppelt so komplex.

In der Praxis verwenden wir den NCSS genau wie schon den CC gewichtet, dass heißt bspw. den Durchschnitt der 10 Klassen mit den höchsten NCSS-Werten.

9.3 Objektorientierte Metriken

Objektorientierte Metriken sind, wie der Name schon vermuten lässt, Metriken über Objektorientierte Konzepte, Vererbung, Felder etc. Sie bieten uns in der Regel eine relativ gute Einschätzung über die intrinsische Qualität unseres Codes.

9.3.1 Weighted Methods per Class (WMC)

Die gewichteten Methoden pro Klasse sind eine Aufsummierung der Komplexitäten aller Methoden einer Klasse. Als Komplexität nehmen wir der Einfachheit halber die Zyklomatische Zahl, obwohl hier auch andere Kenngrößen denkbar wären.

WMC bietet uns Rückschlüsse über die Wartbarkeit einer Klasse, denn sowohl mit vielen kleinen Methoden als auch mit wenigen großen Methoden steigt der WMC spürbar an.

Sinnvolle Betrachtungen für die Zeitachse sind die Durchschnittswerte über alle Klassen.

9.3.2 Depth of Inheritance Tree (DIT)

Die Tiefe des Vererbungsbaumes gibt an, aus wie viele Vorfahren eine Klasse hat. In Java ist das immer mindestens 1 (`java.lang.Object`), unter C++ kann es auch 0 sein.

Der DIT liefert eine Aussage über die Komplexität einer Klasse und damit auch über deren Wiederverwendbarkeit. Je mehr Oberklassen eine Klasse hat, desto mehr Methoden erbt sie. Damit wird sie zugleich auch spezieller weniger Wiederverwendbar.

Gute Werte liegen zwischen 1 und 3.

9.3.3 Number of Children (NOC)

Die Anzahl der Kinder eines Objektes ist die Anzahl der direkten und indirekten Unterklassen dieser Klasse.

Diese Metrik ist ein direktes Maß für die Wichtigkeit der Klasse.

9.3.4 Coupling between Object Classes (CBO)

Der CBO-Wert einer Klasse gibt an, mit wie vielen anderen Klassen diese Klasse gekoppelt ist. Kopplung heißt in diesem Fall, die Klasse greift auf Methoden oder Instanzvariablen der anderen Klasse zu.

Der CBO wirkt sich direkt auf den Wiederverwendbarkeitswert einer Klasse aus. Je höher er liegt, desto mehr zusätzliche Abhängigkeiten bringt diese Klasse mit ein.

9.3.5 Response for a Class (RFC)

RFC gibt an, wie viele Methoden von einer Klasse aus erreicht werden können. Er setzt sich zusammen aus der Anzahl der Methoden der Klasse, plus aller Methoden, die diese Methoden aufrufen usw., also im Endeffekt alle transitiven Methodenaufrufe, die von dieser Klasse ausgehen können.

Diese Metrik macht eine direkte Aussage über die Komplexität der Klasse.

9.3.6 Lack of Cohesion in Methods (LCOM)

Eine sehr nützliche Metrik ist LCOM. Sie überprüft die Kohäsion der Methoden einer Klasse anhand der gemeinsamen Nutzung von Feldern. Dazu wird von jeder Methode jedes Feld bestimmt, auf das die Methode zugreift. Danach werden Paare über alle vorhandenen Methoden gebildet.

Die Anzahl der Methodenpaare, die keine Gemeinsamkeit haben wird von der Anzahl der Paare, die eine haben, abgezogen, wobei negative Werte als 0 gezählt werden.

Betrachten wir dazu ein Beispiel:

```
public class LCOM {  
    private int a;  
    private int b;  
    private int c;  
  
    public void calc1() {  
        c = a + 1;  
    }  
  
    public void calc2() {  
        c = b + 5;  
    }  
  
    public void calc3() {  
        a = 5;  
    }  
  
    public int getA() {  
        return a;  
    }  
  
    public int getB() {  
        return b;  
    }  
  
    public int getC() {  
        return c;  
    }  
}
```

Um den LCOM zu berechnen, erstellen wir jetzt eine Matrix mit den Methoden in Zeilen und Spalten:

	calc2	calc3	getA	getB	getC
calc1	+	+	+	-	+
calc2		-	-	+	+
calc3			+	-	-
getA				-	-
getB					-

Jedes + gibt dabei Methoden an, die gemeinsame Felder nutzen, jedes – disjunkte Methoden. Der LCOM dieser Methode liegt also bei $8 - 7 = 1$.

Wird der LCOM zu hoch, sollte man darüber nachdenken, die Klasse aufzusplitten.

9.3.7 Bewertung

Objektorientierte Metriken sind ein gutes Mittel, um einen Überblick über die Qualität des Codes zu bekommen.

Grundsätzlich interessieren uns bei diesen Metriken eher die Tendenzen, also wie sich der Code im Laufe der Zeit entwickelt hat. Es kann aber auch sinnvoll sein, einige (eher großzügig bemessene) Grenzen zu definieren, die eine Klasse als ungenügend deklarieren können.

Diese Grenzen sollten ggf. von Projekt zu Projekt variiert werden können.

Ein Vorschlag für derartige Grenzen ist es, sich nicht auf eine Metrik abzustützen, sondern eine Reihe von Grenzwerten zu definieren und das Überschreiten von zwei (oder mehr) dieser Grenzen als Verstoß zu werten.

Ein Beispiel:¹

(Eine weitere Metrik wird hier verwendet: Number of Methods (NOM))

- $WMC > 100$
- $CBO > 5$
- $RFC > 100$
- $NOM > 40$
- $RFC > 5 \times NOM$

¹ Vgl. Linda Rosenberg, Ruth Stapko and Al Gallo, Applying object-oriented metrics, November 1999, <http://www.software.org/metrics99/rosenberg.ppt>

Wir könnten die Anzahl der Verstöße gegen den obigen Regelsatz auch wieder als Metrik betrachten (*Violations of Metrics Limits* – VML). Damit wissen wir in einem Refactoring-Zyklus genau, welche Klassen wir als erstes angehen sollten.

9.4 Statische Analyse Tools (Bug Finder)

Eine weitere Metrik, die wir in unseren Projekten einsetzen sollten, ist die Anzahl von Regelverstößen (*Number of Rule Violations* – NRV). Um sie zu bestimmen, gibt es eine Reihe von statischen Code-Analyse Tools, die den Code gegen eine Anzahl von Regeln prüfen - angefangen von Formatierungen und Dokumentation, über gefährliche Konstrukte (`if (b = a)`) bis hin zu unzumutbarer Verwendung von Bibliotheken.

Beispiele für Tools sind *FindBugs*, *PMD*, *Checkstyle* und *Cppcheck*.

9.5 Laufzeit Metriken

Die bisher vorgestellten Metriken sind alle durch statische Analyse bestimmbar, d.h. das Programm muss nicht laufen, sondern lediglich der Sourcecode wird zur Erstellung herangezogen.

Im Folgenden wollen wir einige Metriken betrachten, die erst zur Laufzeit bestimmt werden können.

9.5.1 Testabdeckung

Die Testabdeckung gibt an, die wie viel Code durch unsere Tests tatsächlich durchlaufen wurde, also wie gut unser Code getestet wurde. Die Ergebnisse werden von Paket auf Klassen bis zur Methoden Ebenen aufgeschlüsselt.

Man unterscheidet hier zwischen Pfad- und Zeilenabdeckung.

Testabdeckungs-Metriken liefern uns ein Gefühl dafür, wie gut und umfangreich unser Code getestet wird. Während eine hohe Testabdeckung noch kein Garant für Fehlerfreiheit sein kann, ist eine niedrige Abdeckung ein Zeichen für Fehleranfälligkeit.

Eine sinnvolle Größenordnung ist 80%.

9.5.2 Builddauer

Eine weitere Metrik, die frühzeitig auf Probleme aufmerksam machen kann, ist die Builddauer. Naturgemäß wird diese im Laufe der Zeit immer weiter steigen, da regelmäßig Klassen und Tests dazu kommen.

Auch die Berechnung anderer Metriken kostet natürlich Zeit, die im besten Fall proportional, häufig aber überproportional zur Größe des Projektes ist.

Um diese Problem in den Griff zu bekommen, wird der Build-Prozess geteilt, in einen Basis-Buildprozess, den die Entwickler und der automatische Build nutzen, und einen erweiterten Build, der alle Tests und Metriken vollständig abarbeitet, aber dafür nur nachts läuft.

9.6 Zusammenfassung

Wir haben in diesem Kapitel eine sinnvolle Metriken kennengelernt, mit der wir eine Einschätzung über die Qualität unseres Codes bekommen können.

Der Nachteil dieser Metriken ist, dass sie den Buildprozess verlangsamen, deshalb sollten sie auch nur mit Bedacht eingesetzt werden. Grundsätzlich gilt, dass Metriken nicht nur der bunten Grafiken wegen eingesetzt werden dürfen, sondern immer auch verstanden und ausgewertet werden müssen.

Regel 9-1: Eingesetzte Metriken müssen verstanden sein und regelmäßig ausgewertet werden.

10

Literaturempfehlungen

10 Literaturempfehlungen

Martin Fowler: *Refactoring: Wie Sie das Design vorhandener Software verbessern*, München: Addison-Wesley, 2000

Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading: Addison-Wesley, 1995

Andrew Hunt und David Thomas: *The Pragmatic Programmer*. Boston: Addison-Wesley, 2000

Robert C. Martin: *Agile Software Development: Principles, Patterns, Practices*, Boston: Pearson Education, Inc, 2003

Robert C. Martin: *Clean Code: A Handbook of Agile Software Craftsmanship*, Boston: Pearson Education, Inc, 2008

Steve McConnell: *Code Complete, Second Edition*, Redmond: Microsoft Press, 2004

Diomidis Spinellis: *Code Quality: The Open Source Perspective*, Boston: Pearson Education, Inc, 2006

Gesamtindex

A

Abfrage 5-23
Absätze 7-9
Absichtserklärungen 6-8
Abstrakte Klassen 4-7
Abstraktion 2-8
Abstraktionsebene 5-19
Abstraktions-Methode 5-5
Aggregation 2-11
Änderungsgrund 3-25
Anpassungsfähigkeit 1-5
Antwortzeit 8-19
Argument-Objekte 5-30
Assoziation 2-10
atomar 8-13
Attribut 2-13
Auftraggeber-Sicht 1-9
Ausgabe Parameter 5-29
auskommentieren 6-16
Aussprechbare Namen 4-22

B

Basisklasse 2-14
Bedeutungsvolle Namen 4-5
Benutzerfreundlichkeit 1-5
betriebswirtschaftliche Sicht 1-9
Bindung 2-28
Black-Box-Tests 2-21
Blockgröße 5-12
Boilerplate-Code 2-23
Builddauer 9-11
Builder-Pattern 4-11, 5-27
Buildprozess 6-3, 9-11

C

CamelCase 4-21
CBO Siehe Coupling between Object
Classes
Code Entropy 9-3
Code Smells 1-12
Codehistorien 6-14

Codeoptimierung 8-21
Coding against Metrics 9-3
cohesion Siehe Kohäsion
Command Query Separation 5-32
Comparable 3-13
Contract 2-20
coupling Siehe Kopplung
Coupling between Object Classes 9-7
Cyclomatic Complexity 9-4

D

Daten-Kopien 8-16
Datenkopplung 2-27
Dependency Injection Pattern 2-26
Dependency-Inversion-Principle 3-29
Depth of Inheritance Tree 9-7
Design Patterns 6-8
Diamant-Problem 3-8
DIP Siehe Dependency-Inversion-Principle
DIT Siehe Depth of Inheritance Tree
Dokumentation 6-3
Domänen-Sprache 4-19
Doxygen 6-3
Dyadische Methoden 5-24

E

Effizienz 1-5
encapsulation Siehe Kapselung
encoding 4-24
enge Kopplung 2-24
Event 5-23
Exit-Punkt 5-14, 5-33, 9-5
Expertensysteme 2-6

F

Factory-Methode 4-10
Factory-Pattern 2-26
Flags 5-27
Flaschenhals 8-21
Flexibilität 1-7
Forcierte Kommentare 6-13

Formale Kommentare 6-9
 Formatierung 3-22, 7-3
 Formatter 7-3, 7-16
 freie Kopplung 2-27
 fünf Prinzipien 3-3
 Funktion 5-4
 Funktionale Programmierung 2-4

G

Gefühlte Performance 8-20
 Genauigkeit 1-5
 Getter 2-22
 Großschreibung 4-21

H

Hacker-Sicht 1-6
 Hilfsmethode 5-5
 Holper-Regel 3-7, 5-16
 Hook-Methode 5-11
 HotSpot-Compiler 5-14
 Hrair-Limit 5-7, 5-26

I

Inhaltskopplung 2-24
 innere Kapselung 2-20
 Integrität 1-5
 Interface 3-9, 4-8
 Interfaces 2-9
 Interface-Segregation-Principle 3-18
 Invariante 2-22
 Inversion of Control 2-26
 ISP Siehe Interface-Segregation-Principle
 Iterator 4-13

J

Javadoc 6-3

K

K
 die drei 2-19
 Kapselung 2-19, 5-5
 Kardinalität 2-13
 Keyword-Form 5-16
 Klammer-Kommentare 6-15
 Klarstellungen 6-8
 Klassen 2-8
 Klassengröße 3-22
 Klassenhierarchien 3-4

Kleinschreibung 4-21
 Kohäsion 2-28, 3-22
 Kommentar 6-3
 Kommunikation 7-6
 Komponente 2-13
 Konstruktor 4-10, 5-26
 Kontext 4-11
 Konzept 4-19
 Kopplung 2-24, 3-30
 Korrektheit 1-5
 Kreis-Ellipse-Problem 3-5
 künstliche Intelligenz 2-6

L

Lack of Cohesion in Methods 9-8
 LCOM Siehe Lack of Cohesion in Methods
 Lead 7-9
 Leetspeak 4-26
 Lesbarer Code 6-4
 Lesbarkeit 1-8
 Lifecycle-Methode 5-11
 Lines of Code 3-22, 9-5
 Liskovsches Substitutionsprinzip 3-6, 5-12
 Logische Programmierung 2-5
 lose Kopplung 2-25
 Lösung-Sprache 4-19
 LSP Siehe Liskovsches
 Substitutionsprinzip

M

Mehrfachvererbungen 3-8
 Merkmale
 extrinsisch 1-11
 intrinsisch 1-11
 Methode 5-3
 Methodenargumente 5-21
 Metrik 9-3
 Missverständliche Namen 4-16
 Mix-In 3-15, 4-8
 Monade 5-23
 Monadische Methoden 5-22
 Moore's Law 8-9
 Muttersprache 4-4

N

Nachrichten 2-18
 Nicht-Funktionalen Anforderungen 1-6
 Niladische Methoden 5-22

NOC Siehe Number of Children
Non commenting source statements 3-22
Non Commenting Source Statements 9-6
NRV Siehe Number of Rule Violations
Number of Children 9-7
Number of Rule Violations 9-10
Nutzer-Sicht 1-4

O

Oberklasse 2-14
Objektorientierte Analyse 2-3
Objektorientierte Programmierung 2-6
Objektorientiertes Design 2-3
Objektorientierung 2-3
Observer 4-19
OCP Siehe Open-Closed-Principle
OOA Siehe Objektorientierte Analyse
OOD Siehe Objektorientiertes Design
Open-Closed-Principle 3-27, 5-10
Operation 5-3
Operationen 2-9
Optimierung 8-19
Optimierungsdreieck 8-22
Optimierungspyramide 8-23
Optische Verwechslung 4-22

P

Package 2-17
Parallele Schnittstellen-Hierarchien 3-16
Performance 8-10, 8-19
Polyade 5-26
Polyadische Methoden 5-26
Polymorphie 2-14, 2-15, 3-4, 4-7, 5-4
Portierbarkeit 1-7
Prefix 4-14
Private 2-17
Produkt-Qualität Siehe Softwarequalität,
extern
Professioneller Code 1-4
Programmierer-Sicht 1-7
Programmierrichtlinien 4-21
Projektrichtlinien 4-4
Protected 2-17
Prototypen 2-8
Prozedur 5-4
Prozedurale Programmierung 2-4
Public 2-17

R

Read-Write-Problem 8-13
Rechtliche Hinweise 6-7
Rechtzeitigkeit 8-19
Redundanzen 6-13
Refactoring 9-3, 9-10
Regelverstöße 6-9
Rekursion 5-34
Response for a Class 9-8
RFC Siehe Response for a Class
Robustheit 1-5
Routine 5-4
Rubrik 7-13
Rückgabewert 4-13

S

Schlagzeile 7-8
Schlammzone 3-29
Schleifenzähler 4-12
Schnittstelle 3-10
 Fähigkeits- 3-12
 Hierarchie - 3-10
 Querschnitt 3-12
Schnittstellen
 Client- 3-18
 Hierarchie- 4-7
Schnittstellenkopplung 2-25
Schnittstellen-Methode 5-5
Schreibfehler 4-17
Seiteneffekt 5-4, 5-32
Separation of Concerns 5-17
Setter 2-22
Short-Circuit-Operator 9-5
Sichtbarkeiten 2-16, 2-20
Signatur 5-5
Single-Repsonsibility-Principle 5-17
Single-Responsibility-Principle 3-25, 4-7
Single-Responsiblity-Principle 8-15
Skalierbarkeit 8-20
Slang 4-26
Softwarequalität
 extern 1-6
 intern 1-7
SOLID 3-31
Sourcecode-Qualität Siehe
 Softwarequalität, intern
Sourcecode-Verwaltung 7-4, 7-15
Spezialisierung Siehe Vererbung
SRP Siehe Single-Responsibility-Principle

Statische Analyse Tools 9-10
 Stepdown-Regel 5-19, 5-34
 Style-Guide 4-28
 Suffix 4-14

T

Template-Pattern 5-12
 Testabdeckung 9-11
 Testbarkeit 1-8
 Testfälle 6-19
 Texttauschen 4-17
 TODO 5-10, 6-17
 TO-Satz 5-18
 Trade-Off Optimierung 8-22
 Transformator 5-23
 Triade 5-25
 Triadische Methoden 5-25

U

UML Siehe 2.3.1 Unified Modeling Language
 Ungarische Notation 4-24
 Unified Modeling Language 2-7
 Unterstreichungen 6-9
 Untertitel 7-8
 Unveränderliche Objekte 8-14

V

Verantwortlichkeit 3-22

Vererbung 2-14, 3-4, 5-10
 Verhalten 2-6
 Verständlichkeit 1-8
 Vertrag 2-20
 Violations of Metrics Limits 9-10
 virtuelle Felder 2-23
 Vision 5-19
 Visions-Prinzip 3-24
 VML Siehe Violations of Metrics Limits

W

Wartezeiten 8-9
 Wartungsfreundlichkeit 1-7
 Weighted Methods per Class 9-7
 White-Box-Tests 2-21
 Wiederverwendbarkeit 1-7
 WMC Siehe 8.3.1 Weighted Methods per Class
 Wortpaar 4-20
 Wortspiel 4-26

Z

Zeitungsartikel 7-7
 Zeitungsmetapher 7-7, 7-12
 Zugriffsmethode 2-22
 Zustand 2-6
 Zuverlässigkeit 1-5